2011

Security of the SHA-3 candidates Keccak and Blue Midnight Wish: Zero-sum property

Liliya Andreicheva

Security of the SHA-3 Candidates Keccak and Blue Midnight Wish: Zero-sum Property

by

Liliya Andreicheva

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Computer Science

Supervised by

Stanisław P. Radziszowski

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

June 16, 2011

The thesis “Security of the SHA-3 Candidates Keccak and Blue Midnight Wish: Zero-sum Property” by Liliya Andreicheva has been examined and approved by the following Examination Committee:

Stanisław P. Radziszowski
Professor
Thesis Committee Chair

Edith Hemaspaandra
Professor

Ivona Bezáková
Assistant Professor

Acknowledgments

I am very grateful to my understanding and supporting advisor Stanisław P. Radziszowski for leading me, while doing this research, and for the Ministry of Education and Science of the Republic of Tatarstan, which had provided my scholarship for completing MS in Computer Science program at RIT.

Abstract

Security of the SHA-3 Candidates Keccak and Blue Midnight Wish: Zero-sum Property

Liliya Andreicheva

Supervising Professor: Stanisław P. Radziszowski

The SHA-3 competition for the new cryptographic standard was initiated by National Institute of Standards and Technology (NIST) in 2007. In the following years, the event grew to one of the top areas currently being researched by the CS and cryptographic communities. The first objective of this thesis is to overview, analyse, and critique the SHA-3 competition. The second one is to perform an in-depth study of the security of two candidate hash functions, the finalist Keccak and the second round candidate Blue Midnight Wish. The study shall primarily focus on zero-sum distinguishers. First we attempt to attack reduced versions of these hash functions and see if any vulnerabilities can be detected. This is followed by attacks on their full versions. In the process, a novel approach is utilized in the search of zero-sum distinguishers by employing SAT solvers. We conclude that while such complex attacks can theoretically uncover undesired properties of the two hash functions presented, such attacks are still far from being fully realized due to current limitations in computing power.

Contents

Chapter 1

Introduction

1.1 The Problem

Security, and especially computer security, is one of the most important challenges in the modern world. Computer security can be defined by three simple terms: confidentiality, integrity, and availability. In short, the idea is to protect all resources from unauthorized access and multiple attacks, while keeping them available for all authorized users. This is a rather simple task, in the case of a single individual working on his/her PC, however problems arise the moment an Internet connection is established. Due to direct exposure, all possible threats should be seriously considered in a network environment.

One of the fundamental areas in computer security, which is a way of helping us protect our data and our Internet connection, is cryptography. It contains several sub-areas: symmetric-key cryptography, public-key cryptography, etc. One of the most interesting and growing areas of cryptography is hashing. The hash function standard plays an important role in the problems of the modern security. Cryptographic hash functions are used in many applications, for example, digital signatures, message authentication codes (MACs) and other forms of authentication. In addition, they can be used in indexing hash tables, computing checksums, fingerprinting etc. The choice of a secure hash function is crucial. The current standards SHA-1 and SHA-2 perform well, however, a constant threat exists, for example, the enormous progress in computational abilities of the modern computers and supercomputers, which can disarm some hash functions. The issue here are attacks on the

standards; for example, the significant attack on SHA-1 or the attack on MD-5 presented by Wang in 2005. These facts make the functions unstable or unreliable, which leads to the conclusion that the standards need to be updated and optimized.

In 2007 the National Institute of Standards and Technology (NIST) announced a contest for the new standard, SHA-3. We are currently approaching the final stage of the competition and within the next year the final decision is expected. According to the importance of the area and the current interest on this topic inside the computer security community, we decided to work on this subject for this master's thesis.

1.2 Our Contributions

Before studying in particular about two SHA-3 candidate funtions, we are introducing in general the area. We begin by providing an overview of the history of the SHA-3 competition. The thesis work has several goals. The first objective is to overview, analyse and critique the SHA-3 competition. The second one is to perform an in-depth study of the security of two candidate hash functions, the finalist Keccak and the second round candidate Blue Midnight Wish. The study shall primarily focus on zero-sum distinguishers, which is one of the new ideas in cryptanalysis. First, we attempt to attack reduced versions of these hash functions and see if any vulnerabilities can be detected. This is followed by attacks on their full versions. This research approach allows us to find out how the complexity of search increases and are there any dependencies between the different parameters, that we can vary in the algorithm. In the process, a novel approach is utilized in the search of zero-sum distinguishers by employing SAT solvers.

Finally, we describe the attack on the presented functions, providing the experiments and the achieved results. The attack is presented by the generic algorithm for zero-sum distinguishers. We provide the full description of the algorithm and then the experimental part. We also discuss possible future work on this topic.

1.3 Overview

In Chapter 2 we give general definitions necessary for the understanding of the particular cryptographic research area. Next, we provide an overview of the history of the SHA-3 competition. Chapter 3 describes the background of the SHA-3 competition and gives the overview of the first stages. The finalist of the contest Keccak is presented in Chapter 4, with related research currently performed on this function. Chapter 5 describes another candidate function Blue Midnight Wish in the same manner. In Chapter 6, we discuss the general idea of the attack and the use of different SAT solvers as a tool to complete zero-sum distinguishers attack. Chapter 7 describes the experiments and achieved results. Chapter 8 performs conclusions and possible future work.

Chapter 2

Definitions

2.1 Hash functions

Before we begin the overview of the selected hash functions, we need to specify their definitions and properties. First, we observe the idea behind the hash functions. The easiest comparison will be the fingerprints, which are unique for every human, thus there can not be any repeting values. Hash functions are providing a mechanism of obtaining unique fingerprints of data, with such a property that if the data is changed, the value of the function will be changed.

A *hash function* is a mathematical function, that converts some data into a fixed length string of characters.

A *cryptographic hash function* is usually represented by the algorithm, which will also return a fixed-sized bit-string for a given message, such that any change of the data will be followed by the change of the hash value. The data to be encoded is often called the "message", and the hash value is sometimes called the message digest or digest [43, 47]. Hash functions should have 3 significant properties:

- **Problem 1.** Given a hash function $h : X \to Y$ and an element $y \in Y$. Find $x \in X$ such that $h(x) = y$.

  It is infeasible to find an original message on the given message digest (one-way function or preimage resistance).

The complexity of the brute force attack for finding the solution for this problem is $2^n * f(n)$, where n is the length of the hash and $f(n)$ is the complexity of the hash function.

- **Problem 2.** Given a hash function $h : X \to Y$ and an element $x \in X$. Find $x' \in X$ such that $x' \neq x$ and $h(x') = h(x)$.
  It is infeasible to find a message different from the given message, so that their hashes will be the same (second preimage resistance).
  For this problem the complexity of the exhaustive search is the same as for preimage-resistance.

- **Problem 3.** Given a hash function $h : X \to Y$. Find $x, x' \in X$ such that $x' \neq x$ and $h(x') = h(x)$.
  It is infeasible to find two different messages that hash to the same hash value (collision resistance).
  Due to the birthday paradox the lower bound for the complexity of the attacks on this property is significantly improved and estimated as square root of the time needed for the full brute force.

To guarantee the security of a hash function, designers should control all three aspects mentioned. Why these properties are important? As we will discuss further the applications of hash functions, we will see that they are broadly used in many computer programs to provide security. Thus, we want to avoid such undesirable properties, as for example encoding of different messages to the same value, i.e. collisions. The vivid example of how one can cheat, using the hash functions' collisions, is the experiment done by the group of researches with the prediction about the elections of the US President in 2008. They made secret a prediction about the winner, published the hash value of the document, and claimed that their prediction would be correct. The trick was that the researchers were able to produce a set of documents with names of all the candidates, that would hash to the same value, which was announced. Thus, we see that the used hash function, namely MD5,

was not collision-resistant. This fact clearly showed the danger of further usage of MD5.

At the same time the function should be fast to compute and should use little memory, so it can successfully operate in streaming mode. Besides the main properties, a hash function should be resistant to other popular attacks like zero-preimage, length extension, etc. It is difficult to build up an ideal hash function, meaning resistant in terms of collisions, preimages and second-preimages. With the current computational power, "problems which are infeasible" require more than 2^{80} computational effort, which may remain valid for a number of years. Nevertheless, the scientific progress in the technical area is enormous and new ideas such as quantum computers and their incredible computational abilities are very promising, however this is only a research topic at present.

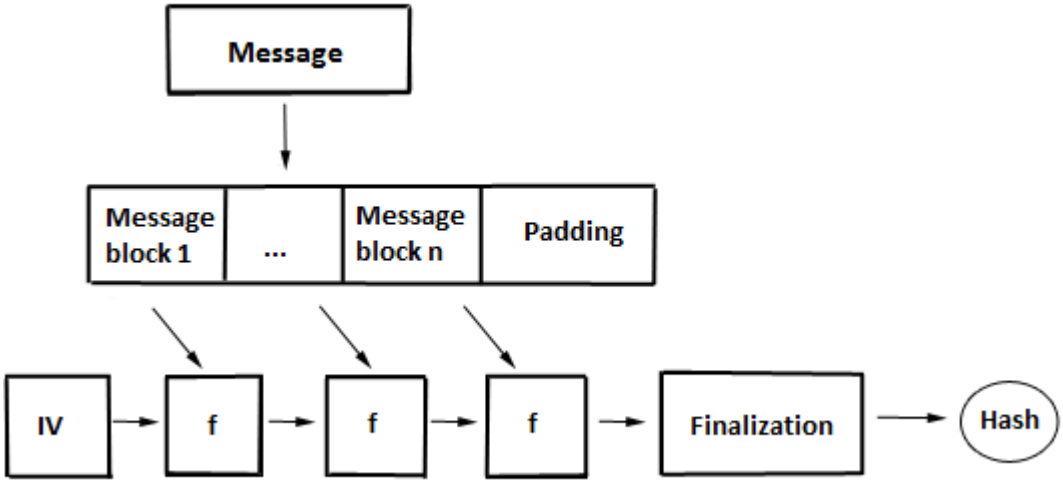
2.2 Merkle-Damgård construction

Let us observe a couple of important design patterns. One of the most widely used architectures of the cryptographic hash functions is Merkle-Damgård construction [43]. As we follow the definition, the hash function should process the input of arbitrary length and produce the output of the fixed size. To implement this mechanism the authors suggested an idea of breaking the input up into a series of equal-sized blocks, and then applying on them the specific compression function, that would be able to deliver the output of the correct size. The compression function is the heart of the hash function, specifically designed in every case.

Now we will take a look on the specifics of the construction itself (Figure 1). First, the message is padded and divided into n blocks, the compression function f takes the input anf the value of the initialization vector (IV). The IV is usually a constant, defined by the designers of the algorithm. Then the compression function processes the input and produces the output of the same size as one of the inputs, that is an intermediate result, which is taken with the next input message block on the next step. After this iterative structure is

processed, to strengthen the construction the last output goes through the finalization.

Figure 1. Merkle-Damgård construction scheme ([47]).



On the basis of Merkle-Damgård construction designers distinguish 2 different approaches: narrow- and wide-pipe. In the wide-pipe construction the second chaining value is added, thus the internal state becomes larger. In this case the length of the hash function output will be twice bigger than the length of the message block and twice bigger that the length of the chaining variable. The idea was first proposed by Stefan Lucks. On Figure 2 you can see the illustration of the idea.

On top of that Mridul Nandi and Souradyuti Paul presented the fast wide-pipe (Figure 3). They managed to make the computation faster by combining the second chaining value with the output of the compression function. In the case of narrow-pipe design the length of the output of the hash function will be the same as the length of the chaining input, but will be twice larger than the length of the message block. Both designs are used in various hash functions, for example, in the SHA-3 competition such functions as JH, Echo, BMW present the wide-pipe construction, while Hamsi, Shavite3, Skein refer to narrow-pipe design.

Figure 2. Wide-pipe design ([47]).

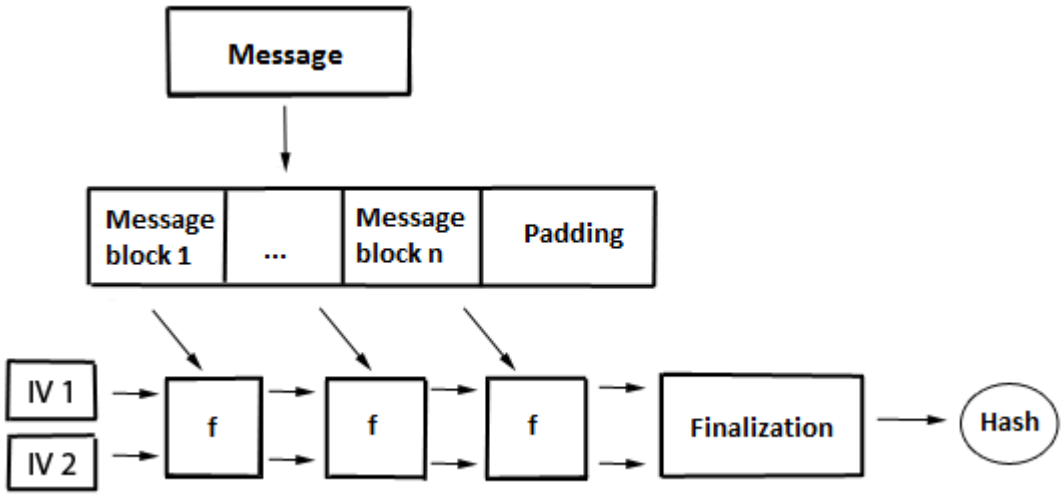
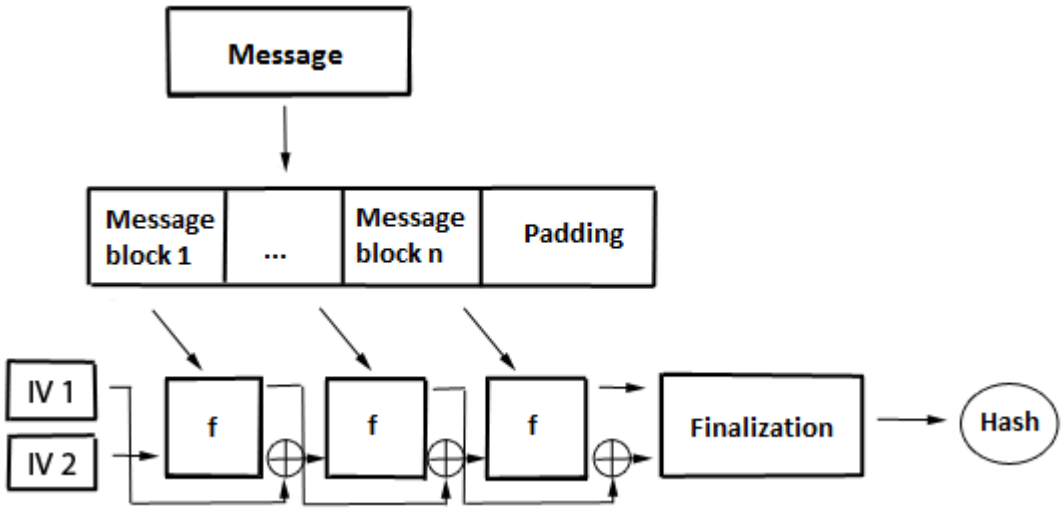


Figure 3. Fast wide-pipe design ([47]).



One of the most famous representatives of Merkle-Damgård construction is MD5, which is the basis for SHA-1. Among the reasons of the popularity of this construction is the proven fact that if the compression function f is collision resistant, then so is the hash function, which uses it [47]. Unfortunately, this construction also has several undesirable vulnerabilities against the following attacks:

- Length extension attacks – given $h(m)$ and $len(m)$, but not m, by choosing a suitable

m', an attacker can calculate $h(m\|m')$, where $\|$ denotes concatenation. Once an attacker has one collision, he can find more very easily.

- Second preimage attacks – given a fixed message m_1, find a different message m_2, such that $h(m_2) = h(m_1)$. This type of attack against long messages are always much more efficient than brute force.

- Multi-collisions attacks – given a hash function $h()$ and a set of messages $\{m_1, ..., m_r\}$, multi-collision or r-multi-collision on a function is a property, such that $h(m_1) = h(m_2) = ... = h(m_r)$. In other words many messages have the same hash value. Antoine Joux was investigating this question a lot and his results show that it is not much harder to build a collision on 2^t messages, than on 2 inputs [20].

2.3 Sponge construction

Sponge construction is a novel type of construction, which is used in the design of hash functions. For example, it is the basic idea in the design of Keccak. Let us look at the main features of this approach.

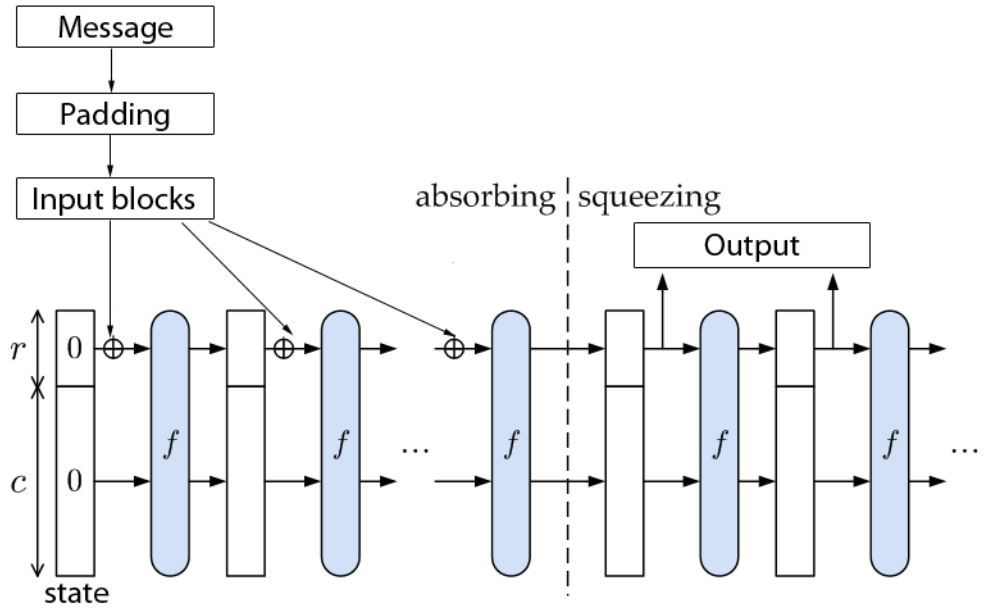
The designers describe 2 main phases of the algorithm: the absorbing phase, which takes the input and calls the compression functions, and the squeezing phase, that delivers the output (Figure 4). The sponge construction works in the iterative manner, depending on how many rounds eventually has the hash function. Several principal attributes should be defined to understand this approach:

- We assume that the compression function f is operating on the fixed number of bits b, that parameter is called the *width*. The width b consists of 2 parts: bitrate r and capacity c, $b = r + c$. We call these b bits of the construction the *state*. Basically, the compression functions takes the bits of the input message and the bits of the state and performs all kinds of permutations and transformations on them.

- The bitrate r is the number of bits, which represent the part of state that is mixed with the same number of bits from the input. The input message is processed in blocks of the size r, and even before the absorbing phase it is padded. Padding rule depends on the particular hash function and usually the size of the output of the hash. After the absorbing phase the first r bits of the state with the function f applied to them are returned as the output to the user.

- The parameter c corresponds to the part of the state, which is never interleaved with the input values, but is is processed with the compression function as well. This attribute is supposed to increase the security level of the construction.

Figure 4. Sponge construction scheme ([10]).



One of the most important criteria of the successful hash function is the level of security. Thus, the designers of different functions and constructions work thoughtfully on the main properties of the hash functions in order to provide high level of preimage, second-preimage and collision resistance. We usually describe the complexity of breaking the function (in terms of breaking one of the 3 properties, listed above) as the number of operations that

are needed to perform the attack. The authors of the sponge construction, claim that the security of this algorithm depends on the capacity and bitrate. These are 2 parameters that vary from the size of output, thus for the particular hash function, based on the sponge construction, we can have different sizes of the output value.Thus, the security claims of the sponge construction with the certain capacity c describe the following cases [10]:

- Collision resistance: if the output length is nc, the collision resistance level is $2^{n/2}$. If the output length is nc, the resistance level is $2^{c/2}$.
- (Second) preimage resistance: If the output length is $nc/2$,, the (second) preimage resistance level is 2^n. If the output length is $nc/2$, the resistance level is $2^{c/2}$.

2.4 Applications

We are using hash functions extensively in our daily work, often without even noticing. Everyone who is working with a computer bumps into hashing in some way. Especially, when you are in the cryptography or programming field. We will describe several applications of hash functions in the modern world.

Today, E-mail and electronic documents are an essential part of our lives. For a long time now the electronic signature has been the method used for document verification, having the same power as a handwritten signature. This concept is implemented via a digital signature. Historically, digital signatures were the first application of cryptographically secure hash functions. Hash functions have a dual role in practical signature schemes: they expand the domain of messages that can be signed by a scheme and they are an essential element of the scheme's security. We have several goals in this case: to prevent any unexpected changes in the content of the message, to be able to prove that message comes from the right person, etc. Thus the most convenient mechanism will be the one, which transforms any signature scheme good for signing messages of a fixed length, and a collision-resistant hash function into a signature scheme, can handle arbitrary-length messages. This ideas

are implemented in so called the hash-and-sign paradigm, that is the basis for all modern practical signature schemes [26]. A vivid example of this concept is how the SHA-1 is a part of Digital Signature Algorithm (DSA) and Elliptic Curve Digital signature Algorithm (ECDSA). In this case hash functions provide us with fast and secure method to keep our documents and mail safe.

Another highly widespread application of hash functions is message-authentication code (MAC). MAC is a keyed hash function, satisfying certain cryptographic properties, which may be used to verify the integrity and authenticity of information [26]. We describe an example in terms of a typical situation for cryptography - conversation between Alice and Bob. Assume that they want to keep privacy of their correspondence. They share a secret key k. Whenever Alice sends a message m to Bob, she appends an authentication tag $v = h(k, m)$. Bob checks the received value by comparing it to the hash value of the message, that he calculates himself. The goal is to prevent the adversary from forging a message-tag pair, that would be accepted by Bob, and fool him into believing that the message was from Alice. As we know properties of the hash functions, we can easily conclude that implementation of this scheme will be perfect through hashing. Collision resistance gives us safety in the sense of getting the original message unchanged, while the high speed of calculating a hash value allows us to easily compute the authentication value.

More than a quarter of the world population uses the Internet - search engines, commercial web-sites, social networks, etc. People have multiple e-mail accounts, profiles in different systems, accounts in various web-stores, and everywhere client authentication is required. The client needs to present a password previously registered with the server. Storing passwords of all users on the server poses an obvious security risk. Fortunately, the server does not need to know the passwords it may store their hashes (together with some salt to frustrate dictionary attacks) and use the information to match it with the hashes of alleged passwords [29]. It is easy to verify a correct hash provided by a client (fast computation of hash value), but it is very difficult for the adversary to reconstruct the original

password from the hash, as the hash functions are one-way functions.

For programmers the most wide-used application of the hash functions is a hash table, which is a data structure that supports efficient store and look-up operations. Also hash algorithms are used as web certificates, pseudo-random number generators, cryptographic protocols as SSL, SSH and many other applications.

Chapter 3

Secure Hash Algorithm history

3.1 Overview of SHA-family

As hash functions are utilized in a variety of applications, the idea of having a standard in hashing becomes apparent. In the beginning of 1990's the decision was made to design a Secure Hash Algorithm (SHA), which was supposed to become a federal standard. Currently, the cryptographic community is in the process of the selection of the forth function which will be good for several decades in future (the ancestors of SHA-3 function were SHA-0, SHA-1 and SHA-2). However, with the rapid growing of computational abilities of modern machines some of these functions were declared as insecure. To understand the new developments in this area, let us examine these functions more carefully.

SHA-0 was issued in 1993 by NIST. It was very soon replaced by slightly improved version called SHA-1. The base of both functions was MD-5 by Ron Rivest from MIT. The difference between these 2 versions was the additional rotational operation [47]. In 1997 the attack on SHA-0 was found which provided a collision with some probability, producing further changes in standards. In 1998 two French researchers, Florent Chabaud and Antoine Joux, presented an attack on SHA-0, where collisions can be found with complexity 2^{61}, they needed only 2^{61} messages instead of 2^{80} to obtain the collision. They were experimenting with different versions of the function, varying the number of rounds. Their result was improved and in 2004 the attack on the full version of SHA-0 gave a collision. The complexity of this computation was about 2^{51} which is quite feasible now. The latest

results by Wang in 2005 allow to get collision in 2^{39} hash operations, which undermines the NIST security standards. In fact this attack by Wang declared MD5 and SHA-0 as completely insecure.

SHA-1 is one of the current hash standards and it is still one of the most widespread hash functions. As we know, one of the most popular applications of the hash functions is authorization certificates, plenty of them are using SHA-1 now. Unfortunately, after attacks in 2005 NIST decided that the modern world need new hash standard with the higher security level [31]. Specifically on SHA-1 the researches were able to attack the reduced version of the function (53 rounds instead of 80) with less than 2^{80} messages, later the successful attack was performed on the full version of SHA-1 with complexity 2^{69}. All this activity has seriously compromised SHA-1.

Soon after such results, the community decided on switching to the SHA-2 for security issues. However, even to this day people are using some protocols based on MD5, which is actually broken. NIST has declared that after 2010 SHA-1 can be used only in hash-based message authentication codes (HMACs), key derivation functions (KDFs), and random number generators (RNGs). Basically, nowadays NIST strongly recommends to use SHA-2 family as the security standard for various applications and protocols [31].

SHA-2 was set up as a new standard in 2001. There is the whole family of functions SHA-224, SHA-256, SHA-384, SHA-512 with different sizes of the digest. The name of the algorithm which is now called SHA-2 is Advanced Encryption Standard. The design of both SHA-1 and SHA-2 (and also MD5) uses Merkle-Damgård construction. SHA-2 provides better security guarantee than SHA-1, but although it is already a standard for a decade, SHA-1 is still very widely used. Now vendors are not quite sure if they need to start working with SHA-2 or wait for the new standard, which is SHA-3, because nevertheless despite all the attacks, there are no collisions found on SHA-1. The best known attacks on SHA-2 nowadays are performed only on the reduced versions of the function: 41 of

the 64 rounds of SHA-256 or 46 of the 80 rounds of SHA-512. Vivid examples of SHA-2 usage will be SHA-256 in the authentication process of Debian Linux sotware packages, SHA-256 and SHA-512 in Unix and Linux secure password hashing [47].

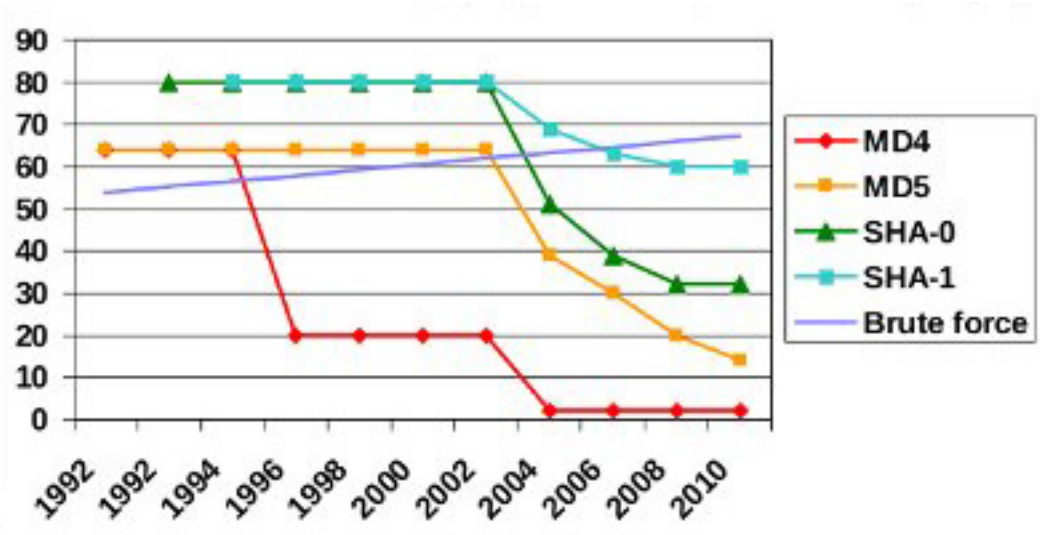
Although the complexity of breaking SHA-2 is still considered infeasible, NIST has decided to start working on the next hash standard, which is basically SHA-3 competition. And the following table presents the comparison of all SHA functions and their characteristics up to this point. Analysing the table, we can see the progress in the development of the hash functions: the latest standard offers the whole variety of versions with different sizes of output blocks, which makes the function more tunable for applications.

Table 1. SHA functions ([47]).

Algorithm	**Output size**	**Internal state size**	**Block size**	**Collisions**
SHA-0	160	160	512	Yes
SHA-1	160	160	512	Yes
SHA-2 (SHA-256/224)	256/224	256	512	None
SHA-2 (SHA-512/384)	512/384	512	1024	None

The following diagram illustrates the progress in the collision attacks within approximately last 20 years.

Figure 5. Complexity of collision attacks ([39]).



The complexities are compared with the standard brute force requirements. As we can observe, the MD family is completely broken nowadays, but SHA-1 still holds the position. If we look precisely at the diagram, we see that SHA-1 is characterized now by the complexities of 2^{60}, which is still hard to compute. Thus it can be used in some applications of cryptographic hash functions, although it is not completely secure now.

3.2 The NIST competition

3.2.1 Background

The competition was launched in 2007. According to the format of the competition, there are several rounds in it. The winner should be chosen in 2012. The submission requirements included such aspects as: free worldwide, implementable on varied hardware and software platforms, must support 224, 256, 384, and 512 bit digests, and messages of at least up to 2^{64} bits. The configuration of the submission package included information concerning possible attack scenarios, source code in ANSI C, time and space requirements for hardware and software for 8-, 32- and 64-bit platforms, issued or pending patents, documentation, and self-evaluation. The decision about candidates for each round is based on the work of the committee and the analysis of the public comments. There is a great amount of work going on around this challenge in the cryptographic world, therefore candidate functions are analysed from different points of view by various researches.

Initially 64 candidate functions were submitted to the committee; the first round in 2008 was dealing with 51 candidates. The next step was the round of public comments and in 2009, 14 participants were approved for the second round. The competition caused a great interest in the cryptographic community, plenty of work is done by individual researches, which helps NIST a lot. A public forum (hash-forum@nist.gov) became quite popular and after a while was declared an official resource, that is used in the evaluation process. As the area is very dynamic and new, there are a lot of comments, notes and essays in the Internet, showing cryptanalysis and performance studies of the candidate functions, but not all of

them can be already found as published papers, although some of them are presented in various conferences [32]. The complete timeline of the event up to the current moment is presented in Table 2 [32]:

Table 2. The official timeline of the SHA-3 competition ([32]).

October 31 - November 1, 2005	Cryptographic Hash Workshop, NIST, Gaithersburg, MD.
August 24-25, 2006	Second Cryptographic Hash Workshop, UCSB, CA.
January 23, 2007	Federal Register Notice - Announcing the Development of New Hash Algorithm(s) for the Revision of Federal Information Processing Standard (FIPS) 180-2, Secure Hash Standard.
November 2, 2007	Federal Register Notice - Announcing a Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. SHA-3 competition began.
October 31, 2008	SHA-3 Submission Deadline.
December 10, 2008	First-round candidates announced. The first round began.
February 25-28, 2009	First SHA-3 Candidate Conference, K.U. Leuven, Belgium.
July 24, 2009	Second-round candidates announced. End of the first round.
September 28, 2009	Second-round candidates posted for public review. The second round began.
August 23-24, 2010	Second SHA-3 Candidate Conference, UCSB, CA.
December 9, 2010	SHA-3 finalists announced. End of the second round.

One of the significant resources that collects information about the competition and its participants is a web-portal SHA-3 Zoo [14], which is hosted by the Graz University of Technology. This web-resource provides lots of materials on the candidate-functions, including original submissions, current publications about the attacks, papers and opinions. An interesting point is that NIST does not provide anything of the kind. Another huge resource is eBash by Daniel Bernstein from the University of Illinois in Chicago,IL [13], which is part of large system benchmarking cryptographic functions. There is a great

amount of data on performance of the SHA-3 candidate functions on various types of platforms [40]. The following table represents the candidate-functions of the second round.

Table 3. The candidate-functions of the second round ([32]).

Hash Name	**Principal submitter**
BLAKE	Jean-Philippe Aumasson
Blue Midnight Wish	Svein Johan Knapskog
CubeHash	Daniel J. Bernstein
ECHO	Henri Gilbert
Fugue	Charanjit S. Jutla
Grøstl	Lars R. Knudsen
Hamsi	Özgül Küçük
JH	Hongjun Wu
Keccak	The Keccak Team
Luffa	Dai Watanabe
Shabal	Jean-Francois Misarsky
SHAvite-3	Orr Dunkelman
SIMD	Gaëtan Leurent
Skein	Bruce Schneier

Although, this set of the functions presents variety of different designs, some of them share the same base constructions. As we can see, there are lots of functions, which are following the popular Merkle-Damgård construction and new sponge design. Table 4 ([21]) by NIST official representative John Kelsey collects the information about the nature of new hash functions.

Table 4. Design diversity ([21]).

	Narrow-Pipe MD		Wide-Pipe MD	
Bitsliced	Hamsi		JH	
AES	Shavite3		Echo	Grøstl
ARX	Skein	BLAKE	BMW	
Logical/ARX			SIMD	Shabal

	Sponge	Sponge-Like
Bitsliced	Keccak	Luffa
AES		Fugue
ARX	Cubehash	

Taking into consideration all current work and research in the cryptographic society, practically all the participants have made some tweaks or adjustments to their original submissions. Here are some of the examples of the latest changes:

- BMW: changed inputs to f_0 and f_1; additional invocation of compression function at the final step.
- CubeHash: double the number of rounds (r=16); increased the size of message block (b=32).
- Hamsi: additional variants specified.
- Keccak: increased message block size (rate); increased number of rounds (from 18 to 24).
- Luffa: modification of S-box; changed order of SubCrumb input; always one round in output transformation.
- SHAvite-3: Inversion of some counter values.
- SIMD: changed rotational constants and permutations for diffusions between parallel Feistels.
- Skein: changed rotational constants.

In fact none of the second-round functions were completely broken. NIST mostly received "partial attacks" - attacks on the reduced, weakened versions of the functions [32]. Eventually, the finalists were announced on December 9, 2010, 3 of the functions are designed by European teams, one comes from the US and one from Singapore and they are:

- BLAKE,
- Grøstl,
- JH,
- Keccak,
- Skein.

In the official comments on the final round NIST states that the committee's primary concerns are taking care of security issues, design and performance of the functions. One of the ideas was to eliminate the functions with similar repeating patterns in the base construction. The authors were allowed to add some tweaks to the functions by January,6 2011. Almost all of them had minor changes shown here:

- BLAKE: final version had an increased number of rounds.
- Grøstl: shift values in one of the permutation were changed; "bigger" round constants are used.
- JH: number of rounds increased.
- Keccak: the padding rule was simplified; the diversifier parameter was removed; the restriction on the bitrate was removed.

Next year, the public comments shall be accepted, and according to the timeline, in 2012 the winner of the competition shall be announced. In fact, NIST will keep analysing the chosen algorithm for 3 more years following the end of the competition. The main goal of such detailed analysis is to obtain the hash standard that will remain secure for at least twenty years after its selection.

3.2.2 Evaluation Criteria

In January, 2007, NIST published a list of the criteria, which candidate-functions should fulfil *Draft Minimum Requirements, Submission Requirements, and Evaluation Criteria for candidate hash algorithms*, all available for public comment in a Federal Register Notice [32, 33]. These requirements and evaluation criteria were updated, based on public feedback, and included in a later, second Federal Register Notice, published on November 2, 2007 [32, 34]. The three main aspects are the security, cost and performance, and algorithm and implementation characteristics.

The main criteria determine if the candidate function will proceed to the next round, but there are several additional issues that are important and can help to decide which of the similar candidates should proceed. One of this additional parameters is the ability to suit equally well to different applications. For example, message authentication codes, pseudo-random number generators, key derivation and one-way functions for obfuscating password files - they all have different requirements, but the hash standard will be used as a part in all of them [41]. Another idea was to make the designers of the functions test the models they invented against particular types of attacks. Here is the list of major security-evaluation factors[32]:

- applications;
- specific requirements for HMAC, Pseudo Random Functions or Randomized Hashing;
- additional security requirements;
- attack resistance.

The second important criteria is cost and performance. NIST has announced the recommended memory consumption and time complexity for the functions. The main goal is to provide better performance of SHA-3 comparing to SHA-2, also taking into consideration

increased security demands. The question of cost is related to the actual implementation in hardware, although the function should provide efficient results in speed as well. In the second evaluation criteria, NIST includes the following points [32]:

- computational efficiency in terms of the speed of the algorithm;
- memory requirements, such as the code size and the random-access memory (RAM) requirements for software implementations and the gate-counts for hardware implementations.

Concerning the third criteria, the algorithm should be *flexible* and *simple* enough, which increases its'chances to be selected to the next round. Numerous new designs were offered, quite different from SHA-2, and the preference was given to the candidates, which can run equally well on different platforms. This reduces cost, and capability of running in parallel increases the performance.

For the functions that have advanced to the second round, additional tweaks to the algorithms were allowed. The reasoning was different. For some functions, adding tunable parameters would help to increase the performance according to the platform and, to others, it was a chance to get rid of some small issues that seemed to be potentially dangerous to the design of the function.

Chapter 4

Keccak

4.1 Specification

Keccak is a family of cryptographic sponge functions created by the team of Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche [11]. The functions differ by the length of permutation. There are 7 Keccak-f permutations, denoted as Keccak-$f[b]$, where $b = 25 \times 2^l$ and l ranges from 0 to 6. Kecccak-$f[b]$ is a permutation over $\mathbb{Z}_2^b$, where the bits are numbered from 0 to b–1. As we know from the definition of sponge construction, b is the width of the permutation. The version with $l = 6$ and $b = 1600$ is one recommended as the main SHA-3 contestant.

The state, which is the main element of the sponge construction, in Keccak is a hypercube a. Basically, we can describe is as a three-dimensional array $a[5][5][w]$, with $w = 2l$, over $GF(2)$. By $a[x][y][z]$, where $x, y \in \mathbb{Z}_5$ and $z \in \mathbb{Z}_w$, we describe the individual bit in position (x, y, z), as coordinates in the hypercube a. If we make a projection of three-dimensional state a to the definition of state in basic sponge construction, we will obtain the following mapping $s[w(5y + x) + z] = a[x][y][z]$. The initial value of the state is the zero-state. The design of the hash function strictly restricts any usage of the initial state as the input, because it can destroy the initial approach. Keccak has a round structure, and each round consists of five functions. Some of them operate on the whole hypercube, but some of them just on parts of it, like slices, rows, columns. Here is the precise description of each round R:

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta.$$

$\theta : a[x][y][z] \leftarrow a[x][y][z] + 4\sum_{y'=0}^{4} a[x1][y'][z] + \sum_{y'=0}^{4} a[x+1][y'][z1];$

$\rho : a[x][y][z] \leftarrow a[x][y][z(t+1)(t+2)/2]$, with t satisfying $0 \leq t < 24$ and

$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^{t} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$ in GF(5)$^{2\times 2}$ or $t = -1$ and $x = y = 0$;

$\pi : a[x][y] \leftarrow a[x'][y']$, with $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}$

$\chi : a[x] \leftarrow a[x] + (a[x+1] + 1)a[x+2],$

$\iota : a \leftarrow a + RC[i_r].$

All the operations are considered in terms of $GF(2)$. Round constants $RC[i_r]$ are predefined and differ from round to round. The number of rounds n_r also varies from the version of the permutation used and is defined by the following formula $n_r = 12 + 2l$ [11].

The heart of the Keccak design is the sponge construction. We denote here the compression function as f, as we know for Keccak f consists of 5 separate functions, m is an input message, standart notation from sponge construction principals bitrate parameter r, capacity c, width b, and padding rule $pad(r)$. The initial state value 0^b is also called the root state. The is a specific restriction, which prohibits to use the value of the root state as input.

1. Generally, we have the sponge function $Z = sponge(m, l)$, where $m \in Z_2^*, l > 0$ and $Z \in Z_2^l$

2. Pad the message m and divide it into blocks of size r, $P = m||pad(r)$, each block is denoted by P_i.

3. Define the root state, $s = 0^b$.

4. For all the input blocks perform compression and interleave with the state - absorbing phase:

$$
\begin{aligned}
&for\ i = 0 to |P|_r - 1\{ \\
&\quad s = s \oplus (|P|_i || 0^{b-r}) \\
&\quad s = f(s) \\
&\}
\end{aligned}
$$

5. Obtain the output bits Z - squeezing phase: $Z = \lfloor s \rfloor_r$

$$
\begin{aligned}
&while\ |Z|_r r < l do\} \\
&\quad s = fs \\
&\quad Z = Z || \lfloor s \rfloor_r \\
&\}
\end{aligned}
$$

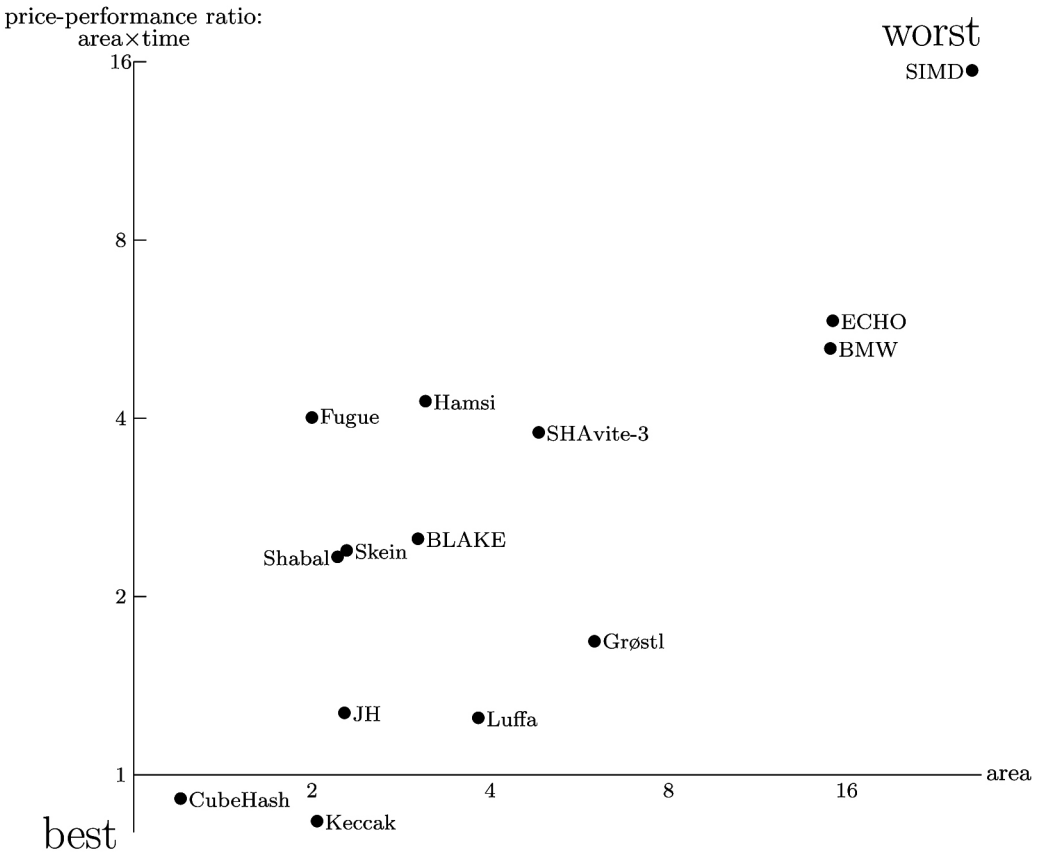
The original version of the function was tweaked after the series of papers about the zero-sum distinguishers by Jean-Philippe Aumasson, Willi Meier and others [4, 8, 12]. In particular, the number of rounds was increased. After the function succeeded to the third round some small corrections were made, you can see the changes in section 3.2. So now, we have the third version of the original submission. The official NIST report says that Keccak was selected as the finalist, mainly, due to the high security level, the simplicity of the design and effective performance[32].

4.2 Performance

One of the important criteria for SHA-3 candidate functions is performance. Still there is no particular standard defined by NIST of how exactly the performance should be measured. Thus, different researches offer various solutions and try as many platforms as possible. One of the successful openings was the System for Unified Performance Evaluation

Related to Cryptographic Operations and Primitives (SUPERCOP) software, developed by the VAMPIRE group with Daniel Bernstein (see appendix B for details). On Figure 6 you can see the diagram, which shows the price-performance ratio of 14 hash functions from the second round of the SHA-3 competition.

Figure 6. Price-performance ratio diagram by D. Bernstein ([5])



This figure shows the comparison of the candidates functions in terms of FPGA platforms. FPGA stands for Field-Programmable Gate Array, kind of programmable curciut, which is widely used nowadays. The vertical axes here is the price-performance ratio, which characterizes the candidates, the lower is the ratio the better it is. The horizontal axes represent FPGA slices, the parts of the curcuit, where agian the smaller the value, the better it is. The diagram shows several result groups: Keccak and CubeHash are best (and

better than SHA-512), with Keccak slightly ahead; JH and Luffa are next; Grøstl is next, only about a factor of 2 behind Keccak and CubeHash; Shabal, Skein, and BLAKE are behind Grøstl; etc. [5]

The easiest way is to measure the performance on the desktop computer, but as there are multiple applications of hash functions, we also need to know the results of performance on FPGA's and so on. Some researches present testing on absolutely non-common platforms such as the Cell Broadband Engine (Cell) and the NVIDIA Graphics Processing Units (GPUs) [7]. In particular in that paper all round 2 functions were tested and Keccak performance was somewhere in the middle among all the non-AES structure candidates, as they were measured separately. Another team was testing 7 functions from the second round on the embedded systems, particularly on smart cards. Thus, Blue Midnight Wish and Shabal are always on top of the rankings, followed by BLAKE, Luffa and Skein. Finally, Grøstl and Keccak are behind. They also studied the tolerance to the passage from 32-bit to 8-bit of each candidates. Overall in these seria of experiments BMW was defined as a leader, followed by Luffa, Keccak and BLAKE, Skein, Shabal, and Grøstl [19]. Some results of the performance on different platforms were collected by John Kelsey for NIST report, we present them in table 5 [21].

In this table we can see 3 types of platforms: ordinary desktop, application-specific integrated circuit (ASIC) and FPGA. They were selected to show the candidates from the different point of views. It is impossible to say which one is the most important, but they all are highly needed in various areas. Thus for ordinary users, who are working with web-sites, for example, every day and thus deal with secure network protocols, the fast performance in software is important (desktop performance). ASICs are integrated circuits designed for very specific needs, thus they represent the significant element of hardware. Thus for the engineers, who design chips and care about their security, it is important to have fast implementation of the hash function in hardware. Programmable circuits –FPGAs are widely used nowadays for numerous applications, even for code breaking, they can

perform massive parallelization, which increases the speed of brute force attack. In order to try these attacks we also need high speed of computation of the hash function itself on FPGAs.

Table 5. Performance of SHA-3 candidates on different platforms ([21]).

NN	Desktop	ASIC throughput	FPGA(ratio)
1	BMW	Luffa	Keccak
2	Shabal	Keccak	CubeHash
3	Skein	CubeHash	Luffa
4	SIMD	Hamsi	JH
5	Luffa	Blake	Grøstl
6	Keccak	Grøstl	Shabal
7	Blake	SHAvite3	Blake
8	JH	JH	Skein
9	CubeHash	BMW	SHAvite3
10	Grøstl	Shabal	Fugue
11	Hamsi	Skein	Hamsi
12	SHAvite3	Echo	BMW
13	Echo	Fugue	Echo
14	Fugue	SIMD	SIMD

From the ranking showed in this table we can conclude that Keccak is overall in the leading group performance-wise. According to the NIST report, Keccak is an average performer in software on most of the platforms specifically for long messages [32]. Although, it is among the top performers in throughput-to-area ratio, which is the hardware parameter. As we can conclude from this table, Keccak performs relatively well on both hardware and software, which gives the function good position in the whole competition. At the same time BMW shows not so stable results.

4.3 Related work

As we have mentioned earlier, most of the analysis on the SHA-3 candidates comes from public comments and research of various cryptography specialists and gurus all over the world. Thus, we have information about very different aspects of the cryptanalysis of the selected function. Nevertheless, none of the attacks performed on Keccak dispute the security guarantees provided by the designers of the hash function.

Let us focus on attacks known as the reduced versions of the function. One of the attacks made by Paweł Morawiecki and Marian Srebrny was implemented with the usage of SAT solver, which is described in their paper [28]. This was a preimage attack only on the 3-round version of Keccak, while the full version contains 24 rounds. Jean-Philippe Aumasson and Dmitry Khovratovich were trying to apply different cube-testers, but analysis of the reduced versions of the function made them think that the full version has quite an adequate number of rounds, which allows to support the initial security claims. These experiments are described fully in the paper [3]. Joel Lathrop was investigating the influence of cube attacks on Keccak [23]. Still, according to his results, cube attacks are successful only for very special cases.

Jean-Philippe Aumasson and Willi Meier presented the so called zero-sum distinguisher in their paper [4], but it doesn't really reflect the security claim of Keccak, as they mention. Authors state that this research is more about investigating how many rounds are needed for Keccak than about disturbing any security claims. Nevertheless, after this paper, the Keccak team decided to increase the number of rounds for Keccak from 18 to 24 in the second round. In [4] we can see deterministic distinguishers for the reduced version of the function with 9 rounds and the authors were also able to get shortcut distinguishers for up to 16 rounds function. After this paper several researches were trying to go deeper into this topic and some of them were rather successful. Thus, Christina Boura and Anne Canteaut in [8] were able to extend the idea of Aumasson and Meier up to 18 rounds of Keccak,

which makes us think that the full version of the function is also not ideal.

The idea of Aumasson and Meier inspired us to continue the research in this direction, thus in this work we are investigating the algorithm for building zero-sum distinguishers. Later in Chapter 6 we give full description of the algorithm and present some new tools for obtaining the solution of the problem. For better understanding of the algorithm, we provide the series of experiments for the hash functions of different size and structure and various dimensions, which finally leads to the tests on the full-size hash functions (see Chapter 7 for details).

Chapter 5

Blue Midnight Wish

5.1 Specification

Blue Midnight Wish (BMW) was designed by Danilo Gligoroski and Vlastimil Klima; the whole team includes also Svein Johan Knapskog, Mohamed El-Hadedy, Jørn Amundsen, Stig Frode Mjølsnes [18, 19]. Gligoroski and Klima were more focused on investigating and improving Turbo SHA-2 hash function, which became the predecessor of BMW. BMW is a family of hash functions, which provides the algorithm for getting the output size of 224, 256, 384 and 512 bits. The function is based on the block cipher approach, the core of the design is wide-pipe Merkle-Damgård construction. The computation runs in 16 rounds, there is an additional parameter P in one of the compression functions, f_1, which defines the number of complex rounds. Consequently, the number of simple rounds is 16 P, default value is $P = 14$.

The general scheme of the algorithm consists the following steps [45]:

1. Preprocessing
    - padding a message,
    - dividing the resulting input into blocks of m –bits,
    - setting the initial values.
2. Hash computation

- generating a message schedule from the padded message,
- generating a series of double pipe values,
- the final double pipe value is used as an input value for a finalization function,
- the n Least Significant Bits (LSB) of the finalization function are used to define the hash value.

The compression function of BMW consists of 3 functions f_0, f_1, f_2. Here is the description of the algorithm provided by the authors, where all the inputs taken by the functions are presented.

Algorithm 1. A generic description of the Blue Midnight Wish hash algorithm ([19])

Input: Message M of length l bits, and the message digest size n.
Output: A message digest $Hash$, that is n bits long.

1. Preprocessing
   - Pad the message M
   - Parse the padded message into N, m-bit message blocks, $M^{(1)}$, $M^{(2)}$, ... ,$M^{(N)}$.
   - Set the initial value of double pipe $H^{(0)}$
2. Hash computation

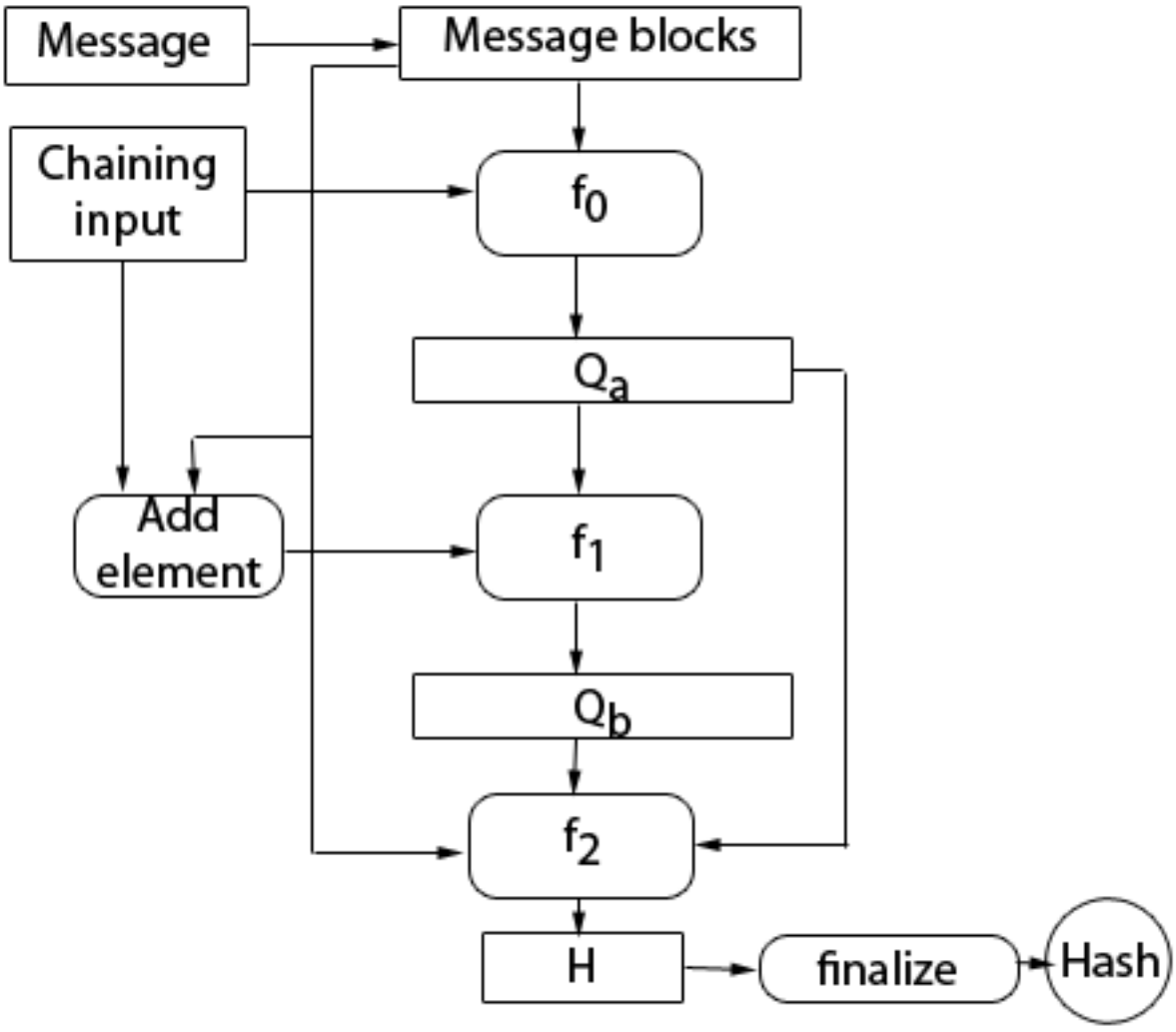
   For i=1 to N
   $Q_a^{(i)} = f_0(M^i, H^{i-1})$;
   $Q_b^{(i)} = f_1(M^i, H^{i-1}, Q_a^{(i)})$;
   $H^i = f_2(M^i, Q_a^{(i)}, Q_b^{(i)})$;
3. Finalization

   $Q_a^{(final)} = f_0(H^N, CONST^{(final)})$;
   $Q_b^{(final)} = f_1(H^N, CONST^{(final)}, Q_a^{(final)})$;
   $H^{final} = f_2(H^N, Q_a^{final)}, Q_b^{(final)})$;

4. $Hash = Take - n - Least - Significant - Bits(H^{(final)})$.

The functions used in the compression function involve a variety of shifts, rotation and XOR operations. There is also a set of predefined constants for the function. All logical functions and constants, that are used in the model of the hash function, can be found in the appendix A. The general scheme of how the compression function operates within BMW, is presented in Figure 6.

Figure 6. Graphical representation of the compression function in BMW ([19]).



Concerning the security questions, the complexity claimed by the designers gives the following statistics:

1. Collision resistance of approximately $n/2$ bits,
2. Preimage resistance of approximately n bits,
3. Second-preimage resistance of approximately nk bits for any message shorter than $2k$ bits,
4. Resistance to length-extension attacks,
5. Resistance to multicollision attacks

Thus, even the smallest complexity for BMW224 for finding collisions yields 2^{112}. The other numbers can be easily computed respectively to the size n of the output of the BMW hash function. We are familiar with the definitions of first three types of vulnerabilities. Let us define the other ones. Length-extension attacks suppose the ability to find message m', if the hash of message m $h(m)$ and the length of m $len(m)$ are given. One of the initial NIST requirements make the designers to define specifically resistance against this type of attack. Multicollision attack is similar to the regular collision, but the only difference is that in multicollision case several message should collide.

The original version of BMW was optimized to prevent the possibility of the attacks described by Soren S.Thomsen [45]. There was a significant tweak, that affected functions f_0 and f_1 and also the finalization round was added. However, the improvements did not stop the attacks, so the security claims were not sufficient enough for the committee. Also, because of the irregularity of the design, which was not understood well enough, Blue Midnight Wish did not proceed to the final round.

5.2 Performance

Comparing to the other participants of the second round of the SHA-3 competition, BMW is a strong performer. It is among the top-performers across most platforms for long and very short messages, according to the official NIST report [32]. Blue Midnight Wish keeps strong positions in speed and average ROM usage as well, although it requires significant amount of RAM, because of its large state size. It is not so good in hardware, but has good potentials for speeding up, which, unfortunately, increases the expenses. The difference in the performance on the software and hardware can be observed in Table 5, which places BMW first for desktop tests and the hardware test results are in the second half of the table.

While talking about the exotic platforms, in [20] the tests on smart cards showed BMW as a leader among other 7 tested functions. In [1] different hardware implementations of BMW were observed. The researchers consider that although BMW does not have the round-like structure, which seems to be a disadvantage in space saving and hardware costs, it is good enough in terms of pipelining. The authors of the function claim BMW to be the fastest hash function among the Second Round SHA-3 candidates.

Overall, NIST characterises BMW as a very good performer with modest memory requirements, which is suitable for a wide range of platforms.

5.3 Related work

The version of BMW which is now in the second round of the NIST competition is a tweaked version. The original one was optimized to prevent the possibility of the attacks described by Soren S. Thomsen. In his paper [44], he showed scenarios for the attacks on the compression function used in BMW. Despite this, the complexities of these attacks applied to the length of output of the hash function are still giving infeasible time ranging from 2^{81} up to 2^{384} for different versions of BMW. All of this made the authors submit

the improved version of the design of the hash function for the second round. Thomsen describes the algorithm how to provide the following attack with the corresponding complexities (where n is the length of the input)[44]:

1. Near-collision attack, 2^{14}
2. Pseudo-collision attack $2^{(3n/8+1)}$
3. Pseudo (second) preimage attack, $2^{(3n/4+1)}$

Near-collision attack aims to find a pair of messages with hash values differing in only few bits. Even if they are found that fact does not seriously affect the security claims of the hash function, because the full collision could never be found. The similar case are the pseudo-attacks, which are the attacks on the compression function, but not on the whole function. That is an interesting question how really the small vulnerabilities in the compression can affect the whole hash, because the complete mechanism of the hash function is usually much more than just use of compression function.

After the tweaked version was presented, the possible attacks are still being developed. J. P. Aumasson presented a practical distinguisher for the compression function of BMW. [30] presents rotational analysis of the hash function and the authors managed to find some weaknesses in compression function of BMW. Another severe attack was presented by Leurent [24]. His attack produced a collision on the first 96 bits of the chaining value with complexity 2^{32}. Their work with Thomsen [25] have extended the attack to 300 of the 512 bits of the chaining value at a similar cost [32].

Nevertheless, the designers of BMW stress the point that BMW is the only hash function in the competition that has only pseudo attacks, thus the security claims still hold. All the authors of the attacks also say that there was not any attack, which affected the security of the function. Table 6 summarizes the significant attacks on Blue Midnight Wish [35].

And as we can see from the table, the number of rounds attacked in one of the compression functions is very small.

Table 6. Significant attack on Blue Midnight Wish ([35]).

Attacker	Hash size	Type of attack	Compression function	
			Attacked variables (rounds)	Compl.
Aumasson	All	pseudo-distinguisher	1 out of 16	2^{19}
Nikolic et.al.	512	pseudo-distinguisher (modified function)	1 out of 16	$2^{278.2}$
Guo and Thomsen	All	pseudo-distinguisher	1 out of 16	2^{1}
Leurent	256	pseudo-collision	3 out of 16	2^{32}
Leurentand Thomsen	256	pseudo-collision	3 full, 7 partial out of 16	2^{32}
Leurent and Thomsen	512	pseudo-collision	3 full, 7 partial out of 16	2^{64}

Chapter 6

Zero-sum attacks with SAT solvers

6.1 The zero-sum property

6.1.1 Definition

The notation of zero-sum distinguisher has been introduced by J.-P. Aumasson and W. Meier in [4]. For a function $f : n \to m$ a zero-sum is a set of inputs $z_1, ..., z_k$, summing to zero in terms of exclusive-or, such that their hashes $f(z_i)$ will also sum to zero, i.e.,

$$\oplus_{i=1}^{k} z_i = \oplus_{i=1}^{k} f(z_i) = 0.$$

As it is considered, there are not many zero-sums for a randomly chosen function, thus several sets of inputs with such a property can be seen as a distinguishing feature of f [8]. How this can be related to the security of hash function? As the zero-sum property is based on the exclusive-or operation, the basic mathematical properties allow us to compute missing input by trivial computation. For example, if we know all inputs in z except one and the f -outputs of all inputs of z but one, we can compute the missing input by xoring all the given data, thus we do not need to call f. If the size of input is small, the attacker will have some advantage in computation, although it does not work properly as the size of input increases [12]. After the work done by Aumasson and Meier in [4], who were able to generate zero-sum distinguishers for the 16-round version of Keccak, Boura and Canteaut followed their ideas and achieved a positive result on the 18-round of Keccak in [8]. The Keccak team has increased the number of rounds after these results were published up to 24. They clearly state the problem in their note in [12], but still the complexity of

computation is so high, that it does not effect the security claims of Keccak.

6.1.2 The generic algorithm

In this work we improve the results given by the generic algorithm by optimizing the calculation of the system of equations. As a tool we use the SAT solver. Let us first present the algorithm and then discuss why SAT solver can be applied. We are going to use the same notations as in [12]:

- N is an integer;
- f is the hash function from n to m bits;
- $X_i = [x_i|f(x_i)]^T$ is a column vector with components the bits of x_i, followed by the bits of $f(x_i)$.

1. Take N random values x_i, compute $f(x_i)$ and form $X_i = [x_i|f(x_i)]^T$.

2. Compute the bitwise sum of the vectors X_i and call the sum A

3. Take $p = n + m + \varepsilon$ random values y_i with $0 \leq i < p$ and ε a small integer, compute $f(y_i)$ and form $Y_i = [y_i|f(y_i)]^T$.

4. Solve the following linear system of $n + m$ equations in the $n + m + \varepsilon$ variables a_i over $GF(2)$, with the bits of $(X_i \oplus Y_i)$ serving as (fixed) coefficients:

$$\Sigma_{0 \leq i < p} a_i (X_i \oplus Y_i) = A.$$

5. And at the last step of the algorithm we form the set Z, such that:

$$z_i = \{y_i, \text{ if } i < p \text{ and } a_i = 1;\ x_i, \text{ otherwise } \}$$

The common way to solve such kind of system is by using Gaussian elimination, but let us examine the efficiency of the solution, with the help of the SAT solver tool. Special feature of SBSAT solver allows to work with equations, containing XOR functions, so we can proceed computing without really changing the structure of equation. However converting these equations to CNF form, will require some time. This method will allow us to obtain the solution much quicker than by calculating it with any traditional algebraic methods.

6.2 Tools

As we have mentioned earlier, we are using SAT solvers as a tool in the discussed algorithm. Although SAT solvers were initially developed to deal with the boolean satisfiability problem, now they represent a very powerful tool, which can be used in a great variety of areas. The following section describes the idea and the current situation in the SAT solvers field, and explains the choice of the SAT solver for our experiment.

6.2.1 Definition

The satisfiability (SAT) is one of the basic problems in classical mathematics and computer science. The goal is to define for a given formula whether the variable can be assigned so that the formula becomes TRUE. The existence of such a solution makes the formula satisfiable. If there is no possible way to evaluate the formula true, the problem is called unsatisfiable. The important thing here is to obtain the certain answer, either positive or negative. Sometimes the problem is mentioned as boolean satisfiability, which stressed the idea of working with bits [47].

SAT was the first known example of an NP-complete problem. That means there is no known algorithm, which efficiently solves all instances of SAT, and it is generally believed that no such algorithm exists [47]. In recent years, some version of SAT problems, such as 2-SAT and 3-SAT were redefined as solvable in polynomial time. Further, a wide range

of other naturally-occurring decision and optimization problems are transformed into instances of SAT. A class of algorithms, called SAT solvers, can efficiently solve a large enough subset of SAT instances useful in various practical areas, such as circuit design, automatic theorem proving and many others. By transforming problems, into SAT instances, we can now solve them with an efficient tool. Extending the capabilities of these algorithms is an ongoing area of research. However, no current such methods can efficiently solve all SAT instances.

Considering cryptographic applications, SAT solvers are efficient enough in attacks on hash function. For example, in [27] the authors analyze the performance of different SAT solvers, which are working on the representation of the attack of Wang on MD and SHA families. At this point SAT solvers appear as an actual tool of cryptanalysis. An important feature is that we are able to turn up the parameters in the SAT solver in such a way that it can be used much more efficiently and faster. Basically, they were capable of producing attacks on MD4 and MD5 in the same manner as the original attacks were with the usage of SAT solvers, though their work was much more time consuming. Another group of researches developed a cryptographic tool, which uses SAT solver, to perform attack on Keccak, one of the SHA-3 finalists [28]. Thus, as we can see from these examples, SAT algorithms represent a powerful approach, which is constantly developing and improving.

6.2.2 SAT competition

Since 2002 a special competition for SAT designers takes place every year, and the best SAT solvers in different categories are selected annually. There are three main classes, in which the competitors can perform: industrial (application), crafted, and random. Numerous designers present different versions of their products in several classes. After the jury of the competition tests the performance of all the solvers on certain amount of instances, the top three performers receive gold, silver or bronze medals. In each of the categories there are also three different classes for unsatisfiable problems (UNSAT), satisfiable problems

(SAT) and all the problems (UNSAT + SAT). There is also a conference, which follows the competition, every couple of years, so that the SAT scientific community keeps the researches up to date. As the organizing committee claims, the main purpose of the whole competition is to identify new challenging benchmarks and to promote new solvers for the propositional satisfiability problem (SAT) as well as to compare them with state-of-the-art solvers [36].

In 2010, the organizers were also accepting solvers that can run the instances in the algebraic format, while usually the SAT solvers are working with so-called DIMACS format. As we know, the instance of the problem to the SAT solver is the boolean formula. So, generally the SAT solvers take as an input the representation in the conjunctive normal form (CNF), which is basically the conjunction of disjunctions of boolean variables. The DIMACS format was developed to easily show the structure of the formula. Let we see the small example.

Example 1. The DIMACS format

c

c this is a comment

c

p cnf 3 2

1 -3 0

2 3 -1 0

This example represents the following boolean formula $(x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_1)$. The CNF file start with comments, these are lines, beginning with the character c. Right after the comments, there is the line ”p cnf nbvar nbclauses”, indicating that the instance is in CNF format; nbvar is the exact number of variables, appearing in the file, and nbclauses is the exact number of clauses in the expression, and then each particular clause is described. The index of the variable is presented as a positive number, if it is just the variable itself in

the clause, and as a negative number if it comes as a negation of the variable in the clause. Each clause ends with symbol of 0. DIMACS format is the most popular format among the SAT solvers.

Another nuance of the recent competitions is that much more attention is given to the parallelization question. Of course, we would like to obtain the solution of the problem no, but it must not take too long. Thus, the organizers decided to promote both the solvers that are quick and those which are more efficient in operation. An important fact is that the organizing committee is keeping the competition alive and growing, as the new tracks and rules are added every year. This ensures progress in the competition.

6.2.3 Some examples

Among the most popular SAT solvers we can name MiniSat. It is one of the first open-source SAT solvers, specifically invented to help researchers and developers. MiniSat is released under the MIT licence, and is currently s basis in the numerous projects. Together with SatELite, MiniSat was recently awarded in the three industrial categories and one of the ”crafted” categories of the SAT 2005 competition [37]. Also MiniSat represents the core of the SAT solver, which we were trying to use at first for developing an attack. The name of the SAT solver is SAT4j and it was released not long ago by one of the organizers of the competition Daniel Le Berre. The main advantage of this particular SAT solver for us is that it is represented as a Java library, which allows for easy implementation in software. This SAT solver is now one of the participants of the SAT competition, so after the conference this year, we can see how good it is among other candidates.

Besides, later while working on the algorithm for the attack, we have found out that SAT4j is not convenient enough for our software. And we started to look for another SAT solver, which can work not only with CNF form, but with other boolean functions. The great solution for my problem was the SBSAT, that is presented in the next section.

In general, all the SAT solvers are well suited for working on the problems of interest, and if performance is not a deterministic factor in the experiments, the winners of the recent SAT competitions are an excellent choice. The following tables represents the results of several last years of the SAT contest.

Table 7. 2010 SAT Race results.

NN	Main track (CNF)	Special track 1 (CNF parallel)	Special track 2 (AIG seq)
1	CryptoMiniSat	plingeling	MiniSat++
2	lingeling	ManySAT 1.5	kw-aig
3	SAT-Power	ManySAT 1.1	NFLSAT

Table 8. 2009 SAT Race results. Application track.

NN	Gold	Silver	Bronze
UNSAT	precosat	glucose	lysat
SAT	SATzilla I	precosat	MXC
ALL	glucose	precosat	lysat

Table 9. 2009 SAT Race results. Crafted track.

NN	Gold	Silver	Bronze
UNSAT	clasp	SATzilla2009-C	IUT-BMB-SAT
SAT	clasp	SApperloT	MXC
ALL	SATzilla2009-C	clasp	IUT-BMB-SAT

Table 10. 2009 SAT Race results. Random track.

NN	Gold	Silver	Bronze
UNSAT	SATzilla2009-R	March hi	NA
SAT	TNM	gNovelty2+	hybridGM3 / adapt2wsat2009++
ALL	March hi	SATzilla2009-R	NA

6.2.4 SBSAT

Our new choice was the SBSAT, which stands for a state-based satisfiability solver, developed and maintained by students and faculty at the University of Cincinnati [38]. The main difference of SBSAT from most of the SAT solvers is that it is working directly with non-CNF input. The authors define the input format as Binary Decision Diagrams (BDDs), which represent complicated boolean formulas using only the operator if-then-else, constants TRUE and FALSE, Boolean variables, and parentheses. They claim that BDD representations are usually much more compact than truth tables [16]. THe other tunable parameter is the search heuristics, which allows to users to experiment, investigating the same problem [15].

In particular the SBSAT is working with the following input formats:

- CNF (Conjunctive Normal Form)
- DNF (Disjunctive Normal Form)
- BDD (SBSAT canonical form)
- Smurf (State Machine Used to Represent Functions)
- Trace
- Prove

- XOR (Each conjoint is an XOR of conjoints)

Some of these formats were developed specifically for the research work on some particular project in the University of Cincinnati, others represent basic forms like CNF, DNF and XOR.

For this work, the ability to take the input in the XOR format was crucial, because of the structure of equations in the chosen algorithm. The translation of these equation to the CNF format by hand would be pretty difficult. Similar observations were made by Paweł Morawiecki and Marian Srebrny in their paper [28] for their SAT-based preimage attack on Keccak, when they had to obtain the CNF for the SAT solver, which they were using. The whole process requires a lot of extra variables, leading to huge memory requirements for real-size cases. The web-page by Mate Soos accurately describes the procedure of converting algebraic normal form to CNF [42]. While choosing the SAT solver, we first tried to work with these conversions and turning the equations in the CNF form. However with the growth of dimensions the problem expanded, so the switch to the SBSAT was made, reducing complexity of the process.

Also a very nice feature of the SBSAT solver is the ability to convert some formats into others, which is convenient for different problems. Here is the table of the input formats conversions [16].

Table 11. SBSAT input format conversion table ([16]).

NN	CNF	DNF	BDD	XOR
CNF	yes	no	no	no
DNF	yes	no	no	no
BDD	yes	no	no	no
XOR	yes	no	no	no

To determine whether format A can be converted to format B, find the format of the input you have in a row. for example, CNF, and the answer appears in the column with the format you want to convert to, for example, XOR. The answer is negative, but if we try to do the opposite thing, switching form XOR to CNF, it can be automatically converted by this type of solver.

The proposed algorithm for zero-sum distinguishers contains the XOR representation of the system of boolean equations. Thus, we used the SBSAT on XOR input. The format of the input is very similar to the DIMACS format for CNF. The header is the following:

p xor max-var-number number of functions ,

where max-var-number defines the number of variables in the formula and number of functions defines the number of equations. Here is an example.

Example 2. The XOR input format
p xor 6 3
x1 x2 x3 = 1
x1x2x3 = 1
x1x2x3 x2x3 x4x5x6 = 0

The equations in this example describe the following formulas:

- $x1x2x3 = 1$ is equal to $equ(xor3(1,2,3),T)$ and $x1 \oplus x2 \oplus x3 = 1$
- $x1x2x3 = 1$ is equal to $equ(and3(1,2,3),T)$ and $x1\&x2\&x3 = 1$
- $x1x2x3x2x3x4x5x6 = 0$ is equal to
  $equ(xor3(and3(1,2,3), and(2,3), and3(4,5,6)), F)$
  and $(x1\&x2\&x3) \oplus (x2\&x3) \oplus (x4\&x5\&x6) = 0$

Overall, we consider the SBSAT the right choice for the proposed algorithm, as it is suited very well to the conditions of the problem and is simple to use.

Chapter 7

Zero-sum experiments

7.1 Part 1. Small case

For the first experiment we are referring the algorithm in [12], which allows to construct a zero-sum distinguisher. We trace the algorithm step by step on the "toy" example. Let assume that $n = 3$ and $f(x) = x + 6$, where $x \in \{0, 2^n\}$.

1. Take N random values x_i, compute $f(x_i)$ and form $X_i = [x_i | f(x_i)]^T$.
   We get the following set of vectors X, where in each vector first 3 bits correspond to the value of x and next 2 bits is the value of $f(x)$:

   0 0 0 1 1 0
   0 0 1 1 1 1
   0 1 0 0 0 0
   0 1 1 0 0 1
   1 0 0 0 1 0
   1 0 1 0 1 1
   1 1 0 1 0 0
   1 1 1 1 0 1

2. Compute the bitwise sum of the vectors X_i and call the sum A:

   $$\Sigma_{0 \le i < N} X_i = A$$

   In our example, A turns into the following vector

$$A = (0\,0\,0\,0\,0\,0)$$

3. Take $p = n + m + \varepsilon$ random values y_i with $0 \leq i < p$ and ε a small integer, compute $f(y_i)$ and form $Y_i = [y_i | f(y_i)]^T$.

   Let assume that $n = 3, m = 3$ and $\varepsilon = 2$, thus $N = p$. And we get p vectors Y:

$$0\,1\,0\,1\,1\,0$$
$$0\,1\,1\,1\,1\,1$$
$$1\,0\,0\,0\,0\,0$$
$$1\,0\,1\,0\,0\,1$$
$$1\,1\,0\,0\,1\,0$$
$$1\,1\,1\,0\,1\,1$$
$$0\,0\,0\,1\,0\,0$$
$$0\,0\,1\,1\,0\,1$$

4. Solve the following linear system of $n + m$ equations in the $n + m + \varepsilon$ variables a_i over $GF(2)$, with the bits of $(X_i \oplus Y_i)$ serving as (fixed) coefficients:

$$\Sigma_{0 \leq i < p} a_i (X_i \oplus Y_i) = A. \tag{1}$$

   Here we compute the bitwise XOR of X_i and Y_i) and store it in the separate vector B, which is in our example equals to

$$0\,1\,0\,0\,0\,0$$
$$0\,1\,0\,0\,0\,0$$
$$1\,1\,0\,0\,0\,0$$
$$1\,1\,0\,0\,0\,0$$
$$0\,1\,0\,0\,0\,0$$
$$0\,1\,0\,0\,0\,0$$
$$1\,1\,0\,0\,0\,0$$
$$1\,1\,0\,0\,0\,0$$

As we have discussed earlier in Chapter 6, the SAT solver shall be used as a tool for obtaining the solution of the system of equations. Let us examine the system, which we can present to the SAT solver later.

Thus, our basic equation is:

$$\Sigma_{0 \leq i < p} a_i B = A.$$

Important to remember is that B_i and A are vectors and a_i are constants, thus we need to take constant a_i as an individual clause with every component of the vector B_i and each chain of clauses equals to the corresponding value in the vector A. Thus we receive the following formula:

$$(a_1 \wedge B_{11}) \oplus (a_2 \wedge B_{21}) \oplus ... \oplus (a_{p-1} \wedge B_{(p-1)1}) = A_1$$
$$...$$
$$(a_1 \wedge B_{1N}) \oplus (a_2 \wedge B_{2N}) \oplus ... \oplus (a_{p-1} \wedge B_{(p-1)N}) = A_N$$

As we are looking for the solution of the whole system of equations, all of the statements should hold true at the same time. Now we need to turn that into special format for the SAT solver. For the input format, which is used in the SAT solver, we need to formulate the equations in terms of the variables. That means we need to look specifically on each closure in every equation of the described system. To simplify the formula we apply some basic rules of discrete math.

- $x \wedge 1 = x$ (2)
- $x \wedge 0 = 0$ (3)

As B_{ij} is a known value, we can apply rules (2) or (3), we get two cases:

(a) $B_{ij} = 0$.

$a_i \wedge B_{ij} = 0$; For XOR statement this case does not matter, as 0 does not influence on the value of XOR.

(b) $B_{ij} = 1$.

$a_i \wedge B_{ij} = a_i$

In this case, we include into the input file the corresponding variable, i.

Processing the whole formula in such a way, we get the file with XOR equations.

The .xor file contains the system of equations in the special input format for SBSAT (see Chapter 6 for details).

p xor 8 6

x3 x4 x7 x8 = 0

x1 x2 x3 x4 x5 x6 x7 x8 = 0

All other equations are trivial, thus they are discarded.

After that file is processed by the SBSAT, we receive the following output:

Satisfiable!

3 (1) val:T

4 (2) val:-

7 (3) val:T

8 (4) val:-

1 (5) val:F

2 (6) val:F

5 (7) val:T

6 (8) val:T

This result gives us the values of the coefficients a_i in our system of equations and they all equal to 1.

5. And at the last step of the algorithm we form the set Z, such that:

$$z_i = \{y_i, \text{ if } i < p \text{ and } a_i = 1; x_i, \text{ otherwise } \}$$

And now we can build the final set:

0 0 0 1 1 0

0 0 1 1 1 1

1 0 0 0 0 0

0 1 1 0 0 1

1 1 0 0 1 0

1 1 1 0 1 1

0 0 0 1 0 0

1 1 1 1 0 1

If we check the XOR of all the final vectors, we receive zero-vector.

At the same time, we can easily receive the unsatisfiable system, for example, for the trivial case, where $X = Y$. The following example is done on the same dimensions, but with other input values:

Table 11. Example 1. Trivial case, $X = Y$

X	Y	$X \oplus Y$
0 0 0 0 0 1	0 0 0 0 0 1	0 0 0 0 0 0
0 0 1 0 1 0	0 0 1 0 1 0	0 0 0 0 0 0
0 1 0 0 1 1	0 1 0 0 1 1	0 0 0 0 0 0
0 1 1 1 0 0	0 1 1 1 0 0	0 0 0 0 0 0
1 0 0 1 0 1	1 0 0 1 0 1	0 0 0 0 0 0
1 0 1 1 1 0	1 0 1 1 1 0	0 0 0 0 0 0
1 1 0 1 1 1	1 1 0 1 1 1	0 0 0 0 0 0
1 1 1 0 0 0	1 1 1 0 0 0	0 0 0 0 0 0

A : 0 0 0 0 0 0

As we see here, the value of $X \oplus Y$ gives a zero-vector, which turns the system of equations to the trivial equality $0 = 0$. That confirms the output of the SBSAT solver, thus, we are observing the trivial case. We can also see the other unsatisfiable case.

Table 12. Example 2. Unsatisfiable case

X	Y	$X \oplus Y$
0 0 0 1 1 0	0 1 0 1 1 0	0 1 0 0 0 0
0 0 1 1 1 1	0 1 1 1 1 1	0 1 0 0 0 0
0 1 0 0 0 0	1 0 0 0 0 0	1 1 0 0 0 0
0 1 1 0 0 1	1 0 1 0 0 1	1 1 0 0 0 0
1 0 0 0 1 0	1 1 0 0 1 0	0 1 0 0 0 0
1 0 1 0 1 1	1 1 1 0 1 1	0 1 0 0 0 0
1 1 0 1 0 0	0 0 0 1 0 0	1 1 0 0 0 0
1 1 1 1 0 1	0 0 1 1 0 1	1 1 0 0 0 0

A : 0 0 0 0 0 0

Solution of the SAT solver (”-” means that the variable can be either 0 or 1): 3 (1) val:F

4 (2) val:-

7 (3) val:T

8 (4) val:-

1 (5) val:F

2 (6) val:F

5 (7) val:F

6 (8) val:F

Z :

0 0 0 1 1 0

0 0 1 1 1 1

0 1 0 0 0 0

0 1 1 0 0 1

1 0 0 0 1 0

1 0 1 0 1 1

0 0 0 1 0 0

1 1 1 1 0 1

Xoring z...

1 1 0 0 0 0

Check of the final vector: false

7.2 Part 2. Sets of small cases

The described algorithm finds some of the solutions that will have an important feature such that:

$$\Sigma_{0 \leq i < N} z_i = 0 \text{ and } \Sigma_{0 \leq i < N} f(z_i) = 0.$$

Basically, the algorithm decides on the given subsets, which input values will be taken in the final solution. If we try all possible inputs of the same dimension, we will most likely receive all solutions. To get the general picture of the whole process, first we try the small samples, which we can easily track for the correctness. Then we do the middle cases, that should show the complexity of the computations. And finally we proceed to the actual hash functions with different variations on the number of rounds and parameters.

Let us first concentrate on how many subsets, which are xoring to 0, we can get working with the vectors of some dimension n. For example, we take different inputs on 3 bits and

the hashing function f generates the output of 3 bits. We need to check the condition that if the subset of input vectors xors to 0, the corresponding subset of hashes of the input vectors also xors to 0. If we want to go through all possible inputs, we receive $2^3 = 8$ vectors of the size 6 bits. To see what subsets will xor to 0, we need to check all possible variants, which is 2^8. As the result we have the following table:

Table 13. Complexity of checking all the subsets for xoring to 0

n	2	3	4	5	6
input size	2^2	2^3	2^4	2^5	2^6
# of subsets	2^4	2^8	2^{16}	2^{32}	2^{64}

As you can see from the table 13, even for relatively small dimensions it is rather time and resource consuming to check all the subsets to get the actual number of solutions. Nevertheless, we calculated some examples, the following table shows the amount of the subsets of vectors, which xor to 0.

Table 14. Number of the subsets, which xor to 0.

$n\|f(x)$	$x+2$	random f_1	random f_2
2	2	1	4
3	10	6	13
4	326	481	520

Comparing the data in the tables 13 and 14, we can conclude that the number of the subsets xoring to 0 is approximately$1/2^{n+1}$. The theoretical probability will be $1/2^n$.

Now, when we have the numerical results of the actual computation, we can proceed with our algorithm, to test for accuracy. We run the algorithm on the different inputs. For this part of experiment we generate the input in the following way:

1. Fix the input X: all the vectors of size n bits. Overall, 2^n vectors.

2. Take some subsets of X as input Y. According to the algorithm size of Y is equal to $p = n + m + \epsilon$, where n- number of the bits in the input vector, m - number of the bits in the hash of the input vector, ϵ - small random number. Let assume that $n = m$, still we have a parameter ϵ to play with. In this experiment we try different amount of vectors in Y, from 2^{n-1} to $2^n - 1$, so that overall the number of variants will be

$$\Sigma C_N^i i!, \text{ where } i = [N/2, N-1] \text{ and } N = 2^n.$$

This formula consists of the number of subsets of i vectors out of N, which is C_N^i, and the number of all permutation inside each subsets, which is $i!$.

3. Take every combination of X and Y as an input. Thus, we get a relatively large amount of inputs for even small dimensions.

After generating all the input files, we run a script, which takes every input file one by one, runs the algorithm on that input, runs the SBSAT for getting the solution of the system of equations, then creates a possible solution from the results received by the SAT solver and checks its correctness. We collect the results of these experiments in table 15.

Table 15. Results of the second part of the experiment.

$n\|f(x)$	$x+2$	random f
2	1	1
3	3	1

The other possible approach for this range of dimensions would be not to fix X or try couple of variations on the size of the vector. Although the dimensions are tiny, when we try all possible combinations, we easily receive a hundred thousand files to process. And as we observe from the character of the growing complexity, for the larger dimensions it is computationally impossible to check all the solutions.

7.3 Part 3. Middle-size cases

The main part of the experiment was to deal with so called “toy”examples, and the main aim of this was to understand the approach and to automate the process of computation. Before we proceed to the samples, which are of the size of the actual hash function, we also want to check the algorithm on several average size examples to obtain the overall statistics. As we can conclude from the part 2 even for small dimension to check all possible combinations of the inputs becomes very difficult, so this will not be performed.

One of the important details in this part is that as we increase the dimension n, we have more possible input vectors out of 2^n scope. That is why we can choose different combinations of vectors in X and Y, so that they do not intersect. Again we are playing with the size parameters, defined by sizes of the input vectors, hashes and ϵ different dimension. According, to these ideas, for the experiments in this part we have tried several approaches. One part of the input samples has followed the pattern $n = m$ and the other was dealing with the cases, where $n < m$ with different values of parameter ϵ and the size of difference between n and m. In similar way as in the main part of the experiment, two “toy”hash functions were used. The dimensions of the samples varied from 8 to 32. Table 16 summarizes the achieved results of this part of the experiment. Values are represented as i/j, where i is the number of solutions for the cases, where $n = m$, and j is the number of solutions, where $n < m$. As discussed above, the exhaustive search of all the solutions was not performed, only random samples on 9 files for the first case ($n = m$) and 18 files for the other one.

Table 16. Results of the third part of the experiment.

$n\|f(x)$	$x + 2$	random f
8	3/0	0/0
16	3/1	0/1
24	5/0	0/0
32	3/1	0/1

As we can conclude from the table above, the more random are the values of the hash-function, the harder it is to find the solution of the problem. On the other hand, for the simplified case, where $n = m$, solutions can be found even when doing relatively small amount of test cases (9 for each dimension).We can see the general tendency in behaviour of this method, but let us proceed to the real-size hashes and hash functions to see how the algorithms works in the real world.

7.4 Part 4. Full-size hash cases

The experiments in the previous parts show that the algorithm is actually working and gives correct solutions. Now we can switch to the full-length hash functions and try different kinds of experiments. In the similar way as in case of middle-size hashes, we work with 2 cases, where $n = m$ and $n < m$. As we know the requirements of the competition suppose 4 standard sizes of the output: 224, 256, 384 and 512 bytes. If we plug in these numbers to the dimensions required by the algorithm, the sizes of the system of equation explode and that makes the SAT solver work for an indefinite period of time. Thus, we are working with just some amount of bytes to represent the value of hash, but taking into consideration all possible hash-function outputs. From this point we receive an extra variation in the experiments, because bytes can be selected from different part of the hash value (least significant bytes or most significant bytes).

At first, we did the experiments on Keccak and then switched to BMW. Now we are going to discuss the results of the experiments and the difficulties encountered.

7.4.1 Keccak

Initially, when we have started working on the algorithm, the small test cases were used to check the correctness of the chosen strategy, but it was done with the so-called “toy”functions. This served as a demonstration to see how the algorithm will work with large inputs, when the actual hash function would be taken for the input. Nevertheless, in

practice this approach was not quite useful. Starting from $n = 24$ bits it is quite difficult to receive the answer from the test program. On $n = 64$ bits the solver is unable to find the solution, the program automatically stops the process. The problem is that with the increase of the size of input, the boolean formula given to the SAT solver is growing progressively. The following table gives an estimate of the size of formula for each tested dimension.

Table 17. Sizes of the boolean formula, given to SBSAT

n	Number of clauses	Number of variables
8	24	30
16	32	41
24	40	48
32	64	64
64	128	128

At this point it is important to mention that we are working with the XOR representation of input, the similar CNF representation of the problem will increase the number of variables for each dimension four times and the number of clauses will increase by *(number of clauses)* $*2^T$, where T is an average number of variables in each clause. Even with these characteristics, running time of the test program for dimensions of $n = 24$, $n = 32$ is up to several hours. Basically, if we estimate n as 2^k, average number of clauses and number of variables is close to 2^{k+1}.

Let us proceed to the organization of the experiment. We have been running the algorithm on different dimensions, specified above. As we can conclude from the previous results, we can not use the whole value, produced by the hash functions, as it yields the large amount of bits. Thus, we decided to take just some subset of bits from the generated hash, which allows us to play with the positions of the bits. Following the general concepts, two variants were chosen to take m most significant bits for one series of test cases and m least significant bits for the other series. As discussed above (Chapter 4), Keccak has round-based structure, that allows us to apply the algorithm to the different versions of

it, from 1 up to all 24 rounds. Overall, the experiments included different variations and combinations of the following parameters:

- dimensions of n and m
- number of rounds in Keccak
- different versions of Keccak with the output size of 224, 256, 384 and 512 bits.
- positions of the bits, takes as the hash value from the full-sized hash
- N number of input vectors

Unfortunately, the algorithm did not show any significant results. While working with the actual hash-function, it was not able to find any distinguishers. These results give credit to the hash function itself and thus does not affect negatively any of the security claims of the authors. As far as explanation of the results of the experiment we see 2 main issues: the tricky structure of the Keccak algorithm, which includes a lot of permutations and constants, and the complicated boolean formula, which represents the system of equations. Another aspect may be the technical characteristics of the testing machine (regular laptop: Intel(R) Core(TM)2 Duo CPU P9300 2.26GHz, RAM: 4Gb, ROM: 320Gb), i.e., some parallelization in SAT solver would provide faster results. All test programs were written in Java, using the code from the original submissions to SHA-3 hash competition for getting the hash values and SBSAT solver as a tool for obtaining the solution of the system of equations (see appendix C for details).

7.4.2 Blue Midnight Wish

The second function, which we are investigating, is Blue Midnight Wish. The experiments for this function were organized in the similar manner as for Keccak, but the important point is that because of the structure of BMW, that does not include rounds in general, only in one part of the compression function, we do not test reduced versions of BMW.

Still BMW is designed to produce 4 different sizes of the output (224, 256, 384 and 512 bits). Even though the design of BMW does not have the repetitive round-structure, one of its' compression functions has a special parameter for so-called complex rounds. Basically, this parameter allows to vary the output, but as BMW has not succeeded for the third round and due to the lack of time, we save this option for the possible future work.

Nevertheless, experiments were performed with such parameters as size of input N and dimensions n and m. Also, as discussed above, the algorithm was tried on both most significant bits and least significant bits. But even with different variations of these parameters, the results were not optimistic. All the results follow one of two possible scenarios:

- The SAT solver says that the boolean formula is unsatisfiable, thus the system of equations does not have a solution.
- The problem is satisfiable, but after the proposed solution is built, part of the vectors are taken from input X, others taken from Y and after that the solution is added by the rest of the vectors from X to match the sizes, the check of the final vector does not xor to zero. That is an incorrect solution.

The third case in this list will be the most wanted variant - a correct solution, but running the algorithm on random samples did not show such a result. There are several possible reasons for this:

- As we discussed above, it is impossible to check all the solutions for the relatively large dimensions,as it requires great computational effort. That is why random samples have been tested, however these samples did not have any solutions.
- The actual number of solutions is very small, as we have estimated earlier, it is approximately $2^{-(n+1)}$, where n is the size of input, thus it is very difficult to find any.
- The proposed method is nondertermetistic, thus it is quite obvious that it does not give a correct solution all the time.

- It might be that the selected functions are strong enough to resist this algorithm, which gives a good characteristic to the designs of the functions.

Thus, we can say that for the Keccak and BMW the proposed algorithm does not find zero-sum distinguishers even on reduced versions, although it is able to find the solutions for other hash functions. Now, we are going to make some final conclusions.

Chapter 8

Conclusions

8.1 Achieved results

The main goal of the thesis was to investigate the selected hash functions and to see if any significant vulnerabilities could be found. We decided to work on such aspect as search of zero-sum distinguishers with the generic algorithm. The proposed algorithm was fully investigated and properly tested on both Keccak and Blue Midnight Wish hash functions. We have modified the algorithm, using SAT solver as a tool. Specifically, we were working with SBSAT, which we consider the most suitable solver for this problem after investigating various popular SAT solvers. Unfortunately, the received results were not so optimistic, although the algorithm was working properly for "toy"examples. Among several possible reasons, we consider the following as the most important ones:

- The proposed algorithm has a very generic structure, which might also reflect its nondeterministic behaviour.
- The designs of the selected hash functions are quite new and mathematically complicated, that makes them resistant to generic algorithms.
- The SAT solver was working very slow for large dimensions.

Thus, based on our experimental results on reduced versions of the functions, we can conclude that this kind of generic algorithm cannot seriously affect the full versions of hash functions, which proves that security claims of Keccak and BMW hold. At the same

time this does not affect negatively the use of distinguishers. The general opinion is that some of the distinguishers are powerful enough, while others just point out some interesting mathematical properties and are not capable of turning into some serious attack. Nevertheless, distinguishers are the important part of the modern cryptanalysis.

8.2 Possible future work

The topic of this thesis is quite new and there might be numerous ways to develop the described approach. For the possible future work, that arises in this area, we present the following ideas:

- Improve the generic algorithm, making it more function specific, which would require serious investigation of each of the compression functions in the particular hash function.
- Improve the performance of the algorithm by the parallelization of the implementation, that will allow to receive the results faster and increase the dimensions of the test examples.
- Try to use another SAT solver, that would be able to work with the boolean formula of the larger sizes in order to increase the dimensions of the inputs.
- Try the same approach on the compression functions of the hash functions, because the vulnerabilities in the compression functions can show the weaknesses of the full hash.

Certainly, there might be other algorithms able to determine zero-sum distinguishers, applicable for the same hash functions that can help in the analysis of the functions. Especially with 5 finalists selected, every detail of analysis is quite important, since a fully secure standard is highly desirable. Based on the current situation among the finalists, Keccak has very high potential to win the competition.

Bibliography

[1] Akin, Abdulkadir and Aysu, Aydin and Ulusel, Onur Can and Savaş, Erkay. Efficient hardware implementations of high throughput SHA-3 candidates keccak, luffa and blue midnight wish for single- and multi-message hashing. Proceedings of the 3rd international conference on Security of information and networks, SIN '10. 2010. http://doi.acm.org/10.1145/1854099.1854135

[2] Liliya Andreicheva. Blue Midnight Wish. 2010. http://www.cs.rit.edu/∼lna5520/Crypto/BMW.pdf

[3] Jean-Philippe Aumasson and Dmitry Khovratovich. First Analysis of Keccak. Comment on the NIST Hash Competition, 2009. http://131002.net/data/papers/AK09.pdf

[4] Jean-Philippe Aumasson and Willi Meier. Zero-sum distinguishers for reduced Keccak-f and for the core functions of Luffa and Hamsi. Comment on the NIST Hash Competition, 2009. http://www.131002.net/data/papers/AM09.pdf

[5] Daniel Bernstein. Visualizing area-time tradeoffs for SHA-3. Notes on the ECRYPT Stream Cipher project (eSTREAM), 2010. http://cr.yp.to/papers.html#bestat

[6] Daniel Bernstein and VAMPIRE lab. Measurements of SHA-3 candidates. http://bench.cr.yp.to/results-sha3.html

[7] Joppe W. Bos, Deian Stefan. Performance Analysis of the SHA-3 Candidates on Exotic Multi-Core Architectures. Cryptographic Hardware and Embedded Systems,

CHES 2010 Lecture Notes in Computer Science, 2010, Volume 6225/2010, 279-293. http://www.springerlink.com/content/9p48014n967455r7/

[8] Christina Boura, Anne Canteaut. A zero-sum property for the Keccak-f permutation with 18 rounds. Proceedings of SAC 2010, LNCS, Springer-Verlag, to appear, 2010. http://www-roc.inria.fr/secret/Anne.Canteaut/Publications/zero_sum.pdf

[9] Joan Daemen, Michaël Peeters, Gilles Van Assche, Guido Bertoni. Keccak sponge function family main document. September 10, 2009. http://keccak.noekeon.org/

[10] Joan Daemen, Michaël Peeters, Gilles Van Assche, Guido Bertoni. Cryptographic sponge functions. 2007. http://sponge.noekeon.org/

[11] Joan Daemen, Michaël Peeters, Gilles Van Assche, Guido Bertoni. Keccak-reference-3.0. January 14, 2011. http://keccak.noekeon.org/

[12] Joan Daemen, Michaël Peeters, Gilles Van Assche, Guido Bertoni. Note on zero-sum distinguishers of Keccak-f. NIST hash forum, 2010. http://keccak.noekeon.org/

[13] eBash: ECRYPT Benchmarking of All Submitted Hashes. http://bench.cr.yp.to/ebash.html

[14] ECRYPT SHA-3 Zoo. http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo

[15] John Franco, Michal Kouril, John Schlipf, Sean Weaver, Michael Dransfield, W. Mark Vanfleet. Function-Complete Lookahead in Support of Efficient SAT Search Heuristics. Journal of Universal Computer Science, vol. 10, no. 12 (2004), 1655-1692, 2004. http://www.jucs.org/jucs_10_12/function_complete_lookahead_in

[16] John Franco, Michal Kouril, Sean Weaver. SBSAT User Manual and Quick Start Guide. SBSAT Version 2.5b-5. January 8, 2007. http://www.cs.uc.edu/~weaversa/SBSAT.html

[17] Danilo Gligoroski, Vlastimil Klima, Svein Johan Knapskog, Mohamed El-Hadedy, Jørn Amundsen, Stig Frode Mjølsnes. Cryptographic Hash Function Blue Midnight Wish. SHA-3 Algorithm Submission. October, 2008. http://people.item.ntnu.no/~danilog/Hash/BMW/Supporting_Documentation/BlueMidnightWishDocumentation.pdf,2009

[18] Danilo Gligoroski, Vlastimil Klima, Svein Johan Knapskog, Mohamed El-Hadedy, Jørn Amundsen, Stig Frode Mjølsnes. Blue Midnight Wish documentation. SHA-3 Round 2 submission. September, 2009. http://ehash.iaik.tugraz.at/wiki/Blue_Midnight_Wish

[19] Mourad Gouicem. Comparison of seven SHA-3 candidates software implementations on smart cards. IACR eprint, Report 2010/531. October, 2010. eprint.iacr.org/2010/531.pdf

[20] Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. Lecture Notes in Computer Science 3152 (2004), 306–316 (CRYPTO 2004). http://web.cecs.pdx.edu/~teshrim/spring06/papers/general\textendashattacks/multi\textendashjoux.pdf

[21] John Kelsey, NIST. SHA-3 Competition Status Update. ECRYPT II, Leuven (BE). 2010. https://www.cosic.esat.kuleuven.be/ecrypt/../johnkelsey-8sept2010.pdf

[22] Vlastimil Klima and Danilo Gligoroski Generic collision attacks on narrow-pipe hash functions faster than birthday paradox, applicable to MDx, SHA-1, SHA-2, and SHA-3 narrow-pipe candidates. IACR eprint, Report 2010/430, 2010. eprint.iacr.org/2010/430.pdf

[23] Joel Lathrop. Cube Attacks on Cryptographic Hash Functions. M.S., ROCHESTER INSTITUTE OF TECHNOLOGY. 2009. http://www.cs.rit.edu/∼jal6806/thesis/thesis.pdf

[24] Gaëtan Leurent. Self-Defence against Fresh Fruit. Crypto 2010 Rump Session. August 17, 2010, http://rump2010.cr.yp.to/c659ebaf681758e01ccf824fd58f3c42.pdf

[25] Gaëtan Leurent and Soren Thomsen. Practical Partial-Collisions on the Compression Function of BMW. 2010. http://www.di.ens.fr/~leurent/files/BMW_Distinguisher.pdf

[26] Ilya Mironov. Hash functions: Theory, attacks, and applications. Microsoft Research, Silicon Valley Campus. November 4, 2005. research.microsoft.com/pubs/64588/hash_survey.pdf

[27] Ilya Mironov, Lintao Zhang. Applications of SAT Solvers to Cryptanalysis of Hash Functions. Microsoft Research, Silicon Valley Campus. Theory and Applications of Satis

ability Testing - SAT 2006, volume 4121 of Lecture Notes in Computer Science, pages 102115. Springer, 2006. research.microsoft.com/en-us/people/mironov/sat-hash.pdf

[28] Paweł Morawiecki and Marian Srebrny. A SAT-based preimage analysis of reduced KECCAK hash functions. Second SHA-3 Candidate Conference, Santa Barbara. August 5, 2010. http://eprint.iacr.org/2010/285.pdf

[29] Robert Morris, Ken Thompson. Password security: A case history. Communications of ACM, vol. 22(11), pp. 594597. 1979.

[30] Ivica Nikolič, Josef Pieprzyk, Przemysław Sokołowski, Ron Steinfeld. Rotational Cryptanalysis of (Modified) Versions of BMW and SIMD. 2010.

https://cryptolux.org/mediawiki/uploads/0/07/Rotational_distinguishers_Nikolic,_Pieprzyk,_Sokolowski,_Steinfeld29.pdf

[31] NIST. http://nist.gov

[32] NIST. Announcing the Development of New Hash Algorithm(s) for the Revision of Federal Information Processing Standard (FIPS) 180-2, Secure Hash Standard, Federal Register. Vol. 72, No. 14. January 23, 2007 http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Jan07.pdf

[33] NIST. Status Report on the Second Round of the SHA-3 Cryptographic Hash Algorith Competition. February, 2011 csrc.nist.gov/publications/nistir/ir7764/nistir-7764.pdf

[34] NIST. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family, Federal Register. Vol. 72, No. 212. November 2, 2007 http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf

[35] Official web-site of Blue Midnight Wish hash function. http://www.q2s.ntnu.no/sha3_nist_competition/start

[36] Official web-site of the SAT competition. http://www.satcompetition.org/

[37] Official web-site of MiniSat. http://minisat.se/

[38] Official web-site of SBSAT. http://www.cs.uc.edu/~weaversa/SBSAT.html

[39] Bart Preneel. Hash functions: theory and practice. Asiacrypt 2010, ICICS 2010 and Indocrypt 2010 2010. http://homes.esat.kuleuven.be/~preneel/

[40] SHA-3 finalists. Stanisław P. Radziszowski. March, 2011. http://www.cs.rit.edu/~spr/gdn2010/

[41] Andrew Regenscheid, Ray Perlner, Shu-jen Chang, John Kelsey, Mridul Nandi, Souradyuti Paul. Status Report on the First Round of the SHA-3 Cryptographic Hash Algorithm Competition. September, 2009. http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/documents/sha3_NISTIR7620.pdf

[42] Mate Soos. Homepage. ANF to CNF conversion. http://www.msoos.org/anf-to-cnf-conversion

[43] Douglas R. Stinson. Cryptography: theory and practice. Third edition. CRC Press. 2006.

[44] Soren Thomsen. Pseudocryptanalysis of the Original Blue Midnight Wish. Fast Software Encryption 2010, Proceedings, volume 6147 of Lecture Notes in Computer Science, pages 304-317. Springer, 2010. http://www.springerlink.com/content/05663v32l1k34181/

[45] Soren Thomsen. A near collision attack on the Blue Midnight Wish compression function. 2008. http://www2.mat.dtu.dk/people/S.Thomsen/bmw/nc-compress.pdf

[46] Xiaoyun Wang, Yiqun Lisa Yin and Hongbo Yu. Finding Collisions in the Full SHA-1. In CRYPTO'05, volume 3621 of Lecture Notes in Computer Science, pages 1736. Springer, 2005. people.csail.mit.edu/yiqun/SHA1AttackProceedingVersion.pdf

[47] Wikipedia. http://wikipedia.org

Appendix A

Blue Midnight Wish

Here we provide the logical functions and constants, that are used in the compression functions of Blue Mignight Wish hash function.

Figure 8. The design of f_0 compression function

$f_0 : \{0,1\}^{2m} \rightarrow \{0,1\}^m$

Input: Message block $M^{(i)} = (M_0^{(i)}, M_1^{(i)}, \ldots, M_{15}^{(i)})$, and the previous double pipe $H^{(i-1)} = (H_0^{(i-1)}, H_1^{(i-1)}, \ldots, H_{15}^{(i-1)})$.
Output: First part of the quadruple pipe $Q_a^{(i)} = (Q_0^{(i)}, Q_1^{(i)}, \ldots, Q_{15}^{(i)})$.

1. Bijective transform of $M^{(i)} \oplus H^{(i-1)}$:

$$
\begin{array}{lcccccccccc}
W_0^{(i)} &=& (M_5^{(i)} \oplus H_5^{(i-1)}) &-& (M_7^{(i)} \oplus H_7^{(i-1)}) &+& (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) &+& (M_{13}^{(i)} \oplus H_{13}^{(i-1)}) &+& (M_{14}^{(i)} \oplus H_{14}^{(i-1)}) \\
W_1^{(i)} &=& (M_6^{(i)} \oplus H_6^{(i-1)}) &-& (M_8^{(i)} \oplus H_8^{(i-1)}) &+& (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) &+& (M_{14}^{(i)} \oplus H_{14}^{(i-1)}) &-& (M_{15}^{(i)} \oplus H_{15}^{(i-1)}) \\
W_2^{(i)} &=& (M_0^{(i)} \oplus H_0^{(i-1)}) &+& (M_7^{(i)} \oplus H_7^{(i-1)}) &+& (M_9^{(i)} \oplus H_9^{(i-1)}) &-& (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) &+& (M_{15}^{(i)} \oplus H_{15}^{(i-1)}) \\
W_3^{(i)} &=& (M_0^{(i)} \oplus H_0^{(i-1)}) &-& (M_1^{(i)} \oplus H_1^{(i-1)}) &+& (M_8^{(i)} \oplus H_8^{(i-1)}) &-& (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) &+& (M_{13}^{(i)} \oplus H_{13}^{(i-1)}) \\
W_4^{(i)} &=& (M_1^{(i)} \oplus H_1^{(i-1)}) &+& (M_2^{(i)} \oplus H_2^{(i-1)}) &+& (M_9^{(i)} \oplus H_9^{(i-1)}) &-& (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) &-& (M_{14}^{(i)} \oplus H_{14}^{(i-1)}) \\
W_5^{(i)} &=& (M_3^{(i)} \oplus H_3^{(i-1)}) &-& (M_2^{(i)} \oplus H_2^{(i-1)}) &+& (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) &-& (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) &+& (M_{15}^{(i)} \oplus H_{15}^{(i-1)}) \\
W_6^{(i)} &=& (M_4^{(i)} \oplus H_4^{(i-1)}) &-& (M_0^{(i)} \oplus H_0^{(i-1)}) &-& (M_3^{(i)} \oplus H_3^{(i-1)}) &-& (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) &+& (M_{13}^{(i)} \oplus H_{13}^{(i-1)}) \\
W_7^{(i)} &=& (M_1^{(i)} \oplus H_1^{(i-1)}) &-& (M_4^{(i)} \oplus H_4^{(i-1)}) &-& (M_5^{(i)} \oplus H_5^{(i-1)}) &-& (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) &-& (M_{14}^{(i)} \oplus H_{14}^{(i-1)}) \\
W_8^{(i)} &=& (M_2^{(i)} \oplus H_2^{(i-1)}) &-& (M_5^{(i)} \oplus H_5^{(i-1)}) &-& (M_6^{(i)} \oplus H_6^{(i-1)}) &+& (M_{13}^{(i)} \oplus H_{13}^{(i-1)}) &-& (M_{15}^{(i)} \oplus H_{15}^{(i-1)}) \\
W_9^{(i)} &=& (M_0^{(i)} \oplus H_0^{(i-1)}) &-& (M_3^{(i)} \oplus H_3^{(i-1)}) &+& (M_6^{(i)} \oplus H_6^{(i-1)}) &-& (M_7^{(i)} \oplus H_7^{(i-1)}) &+& (M_{14}^{(i)} \oplus H_{14}^{(i-1)}) \\
W_{10}^{(i)} &=& (M_8^{(i)} \oplus H_8^{(i-1)}) &-& (M_1^{(i)} \oplus H_1^{(i-1)}) &-& (M_4^{(i)} \oplus H_4^{(i-1)}) &-& (M_7^{(i)} \oplus H_7^{(i-1)}) &+& (M_{15}^{(i)} \oplus H_{15}^{(i-1)}) \\
W_{11}^{(i)} &=& (M_8^{(i)} \oplus H_8^{(i-1)}) &-& (M_0^{(i)} \oplus H_0^{(i-1)}) &-& (M_2^{(i)} \oplus H_2^{(i-1)}) &-& (M_5^{(i)} \oplus H_5^{(i-1)}) &+& (M_9^{(i)} \oplus H_9^{(i-1)}) \\
W_{12}^{(i)} &=& (M_1^{(i)} \oplus H_1^{(i-1)}) &+& (M_3^{(i)} \oplus H_3^{(i-1)}) &-& (M_6^{(i)} \oplus H_6^{(i-1)}) &-& (M_9^{(i)} \oplus H_9^{(i-1)}) &+& (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) \\
W_{13}^{(i)} &=& (M_2^{(i)} \oplus H_2^{(i-1)}) &+& (M_4^{(i)} \oplus H_4^{(i-1)}) &+& (M_7^{(i)} \oplus H_7^{(i-1)}) &+& (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) &+& (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) \\
W_{14}^{(i)} &=& (M_3^{(i)} \oplus H_3^{(i-1)}) &-& (M_5^{(i)} \oplus H_5^{(i-1)}) &+& (M_8^{(i)} \oplus H_8^{(i-1)}) &-& (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) &-& (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) \\
W_{15}^{(i)} &=& (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) &-& (M_4^{(i)} \oplus H_4^{(i-1)}) &-& (M_6^{(i)} \oplus H_6^{(i-1)}) &-& (M_9^{(i)} \oplus H_9^{(i-1)}) &+& (M_{13}^{(i)} \oplus H_{13}^{(i-1)})
\end{array}
$$

2. Further bijective transform of $W_j^{(i)}, j = 0, \ldots, 15$:

$$
\begin{array}{llll}
Q_0^{(i)} = s_0(W_0^{(i)}) + H_1^{(i-1)}; & Q_1^{(i)} = s_1(W_1^{(i)}) + H_2^{(i-1)}; & Q_2^{(i)} = s_2(W_2^{(i)}) + H_3^{(i-1)}; & Q_3^{(i)} = s_3(W_3^{(i)}) + H_4^{(i-1)}; \\
Q_4^{(i)} = s_4(W_4^{(i)}) + H_5^{(i-1)}; & Q_5^{(i)} = s_0(W_5^{(i)}) + H_6^{(i-1)}; & Q_6^{(i)} = s_1(W_6^{(i)}) + H_7^{(i-1)}; & Q_7^{(i)} = s_2(W_7^{(i)}) + H_8^{(i-1)}; \\
Q_8^{(i)} = s_3(W_8^{(i)}) + H_9^{(i-1)}; & Q_9^{(i)} = s_4(W_9^{(i)}) + H_{10}^{(i-1)}; & Q_{10}^{(i)} = s_0(W_{10}^{(i)}) + H_{11}^{(i-1)}; & Q_{11}^{(i)} = s_1(W_{11}^{(i)}) + H_{12}^{(i-1)}; \\
Q_{12}^{(i)} = s_2(W_{12}^{(i)}) + H_{13}^{(i-1)}; & Q_{13}^{(i)} = s_3(W_{13}^{(i)}) + H_{14}^{(i-1)}; & Q_{14}^{(i)} = s_4(W_{14}^{(i)}) + H_{15}^{(i-1)}; & Q_{15}^{(i)} = s_0(W_{15}^{(i)}) + H_0^{(i-1)};
\end{array}
$$

Figure 9. The design of f_1 compression function

$f_1 : \{0,1\}^{3m} \rightarrow \{0,1\}^m$
Input: Message block $M^{(i)} = (M_0^{(i)}, M_1^{(i)}, \ldots, M_{15}^{(i)})$, the previous double pipe $H^{(i-1)} = (H_0^{(i-1)}, H_1^{(i-1)}, \ldots, H_{15}^{(i-1)})$ and the first part of the quadruple pipe $Q_a^{(i)} = (Q_0^{(i)}, Q_1^{(i)}, \ldots, Q_{15}^{(i)})$. **Output:** Second part of the quadruple pipe $Q_b^{(i)} = (Q_{16}^{(i)}, Q_{17}^{(i)}, \ldots, Q_{31}^{(i)})$.
1. Double pipe expansion according to the tunable parameters $ExpandRounds_1$ and $ExpandRounds_2$. **1.1** For $ii = 0$ to $ExpandRounds_1 - 1$ $Q_{ii+16}^{(i)} = expand_1(ii + 16)$ **1.2** For $ii = ExpandRounds_1$ to $ExpandRounds_1 + ExpandRounds_2 - 1$ $Q_{ii+16}^{(i)} = expand_2(ii + 16)$

Figure 10. The design of f_2 compression function

Folding $f_2 : \{0,1\}^{3m} \rightarrow \{0,1\}^m$

Input: Message block $M^{(i)} = (M_0^{(i)}, M_1^{(i)}, \ldots, M_{15}^{(i)})$,
quadruple pipe $Q^{(i)} = (Q_0^{(i)}, Q_1^{(i)}, \ldots, Q_{15}^{(i)}, Q_{16}^{(i)}, \ldots, Q_{31}^{(i)})$.
Output: New double pipe $H^{(i)} = (H_0^{(i)}, H_1^{(i)}, \ldots, H_{15}^{(i)})$.

1. Compute the cumulative temporary variables XL and XH.

$$\begin{aligned} XL &= Q_{16}^{(i)} \oplus Q_{17}^{(i)} \oplus \ldots \oplus Q_{23}^{(i)} \\ XH &= XL \oplus Q_{24}^{(i)} \oplus Q_{25}^{(i)} \oplus \ldots \oplus Q_{31}^{(i)} \end{aligned}$$

2. Compute the new double pipe $H^{(i)}$:

$$\begin{aligned}
H_0^{(i)} &= \left(SHL^5(XH) \oplus SHR^5(Q_{16}^{(i)}) \oplus M_0^{(i)}\right) + \left(XL \oplus Q_{24}^{(i)} \oplus Q_0^{(i)}\right) \\
H_1^{(i)} &= \left(SHR^7(XH) \oplus SHL^8(Q_{17}^{(i)}) \oplus M_1^{(i)}\right) + \left(XL \oplus Q_{25}^{(i)} \oplus Q_1^{(i)}\right) \\
H_2^{(i)} &= \left(SHR^5(XH) \oplus SHL^5(Q_{18}^{(i)}) \oplus M_2^{(i)}\right) + \left(XL \oplus Q_{26}^{(i)} \oplus Q_2^{(i)}\right) \\
H_3^{(i)} &= \left(SHR^1(XH) \oplus SHL^5(Q_{19}^{(i)}) \oplus M_3^{(i)}\right) + \left(XL \oplus Q_{27}^{(i)} \oplus Q_3^{(i)}\right) \\
H_4^{(i)} &= \left(SHR^3(XH) \oplus Q_{20}^{(i)} \oplus M_4^{(i)}\right) + \left(XL \oplus Q_{28}^{(i)} \oplus Q_4^{(i)}\right) \\
H_5^{(i)} &= \left(SHL^6(XH) \oplus SHR^6(Q_{21}^{(i)}) \oplus M_5^{(i)}\right) + \left(XL \oplus Q_{29}^{(i)} \oplus Q_5^{(i)}\right) \\
H_6^{(i)} &= \left(SHR^4(XH) \oplus SHL^6(Q_{22}^{(i)}) \oplus M_6^{(i)}\right) + \left(XL \oplus Q_{30}^{(i)} \oplus Q_6^{(i)}\right) \\
H_7^{(i)} &= \left(SHR^{11}(XH) \oplus SHL^2(Q_{23}^{(i)}) \oplus M_7^{(i)}\right) + \left(XL \oplus Q_{31}^{(i)} \oplus Q_7^{(i)}\right) \\
H_8^{(i)} &= ROTL^9(H_4^{(i)}) + \left(XH \oplus Q_{24}^{(i)} \oplus M_8^{(i)}\right) + \left(SHL^8(XL) \oplus Q_{23}^{(i)} \oplus Q_8^{(i)}\right) \\
H_9^{(i)} &= ROTL^{10}(H_5^{(i)}) + \left(XH \oplus Q_{25}^{(i)} \oplus M_9^{(i)}\right) + \left(SHR^6(XL) \oplus Q_{16}^{(i)} \oplus Q_9^{(i)}\right) \\
H_{10}^{(i)} &= ROTL^{11}(H_6^{(i)}) + \left(XH \oplus Q_{26}^{(i)} \oplus M_{10}^{(i)}\right) + \left(SHL^6(XL) \oplus Q_{17}^{(i)} \oplus Q_{10}^{(i)}\right) \\
H_{11}^{(i)} &= ROTL^{12}(H_7^{(i)}) + \left(XH \oplus Q_{27}^{(i)} \oplus M_{11}^{(i)}\right) + \left(SHL^4(XL) \oplus Q_{18}^{(i)} \oplus Q_{11}^{(i)}\right) \\
H_{12}^{(i)} &= ROTL^{13}(H_0^{(i)}) + \left(XH \oplus Q_{28}^{(i)} \oplus M_{12}^{(i)}\right) + \left(SHR^3(XL) \oplus Q_{19}^{(i)} \oplus Q_{12}^{(i)}\right) \\
H_{13}^{(i)} &= ROTL^{14}(H_1^{(i)}) + \left(XH \oplus Q_{29}^{(i)} \oplus M_{13}^{(i)}\right) + \left(SHR^4(XL) \oplus Q_{20}^{(i)} \oplus Q_{13}^{(i)}\right) \\
H_{14}^{(i)} &= ROTL^{15}(H_2^{(i)}) + \left(XH \oplus Q_{30}^{(i)} \oplus M_{14}^{(i)}\right) + \left(SHR^7(XL) \oplus Q_{21}^{(i)} \oplus Q_{14}^{(i)}\right) \\
H_{15}^{(i)} &= ROTL^{16}(H_3^{(i)}) + \left(XH \oplus Q_{31}^{(i)} \oplus M_{15}^{(i)}\right) + \left(SHR^2(XL) \oplus Q_{22}^{(i)} \oplus Q_{15}^{(i)}\right)
\end{aligned}$$

Figure 11. Logical functions used in BMW

BMW224/BMW256	BMW384/BMW512
$s_0(x) = SHR^1(x) \oplus SHL^3(x) \oplus ROTL^4(x) \oplus ROTL^{19}(x)$	$s_0(x) = SHR^1(x) \oplus SHL^3(x) \oplus ROTL^4(x) \oplus ROTL^{37}(x)$
$s_1(x) = SHR^1(x) \oplus SHL^2(x) \oplus ROTL^8(x) \oplus ROTL^{23}(x)$	$s_1(x) = SHR^1(x) \oplus SHL^2(x) \oplus ROTL^{13}(x) \oplus ROTL^{43}(x)$
$s_2(x) = SHR^2(x) \oplus SHL^1(x) \oplus ROTL^{12}(x) \oplus ROTL^{25}(x)$	$s_2(x) = SHR^2(x) \oplus SHL^1(x) \oplus ROTL^{19}(x) \oplus ROTL^{53}(x)$
$s_3(x) = SHR^2(x) \oplus SHL^2(x) \oplus ROTL^{15}(x) \oplus ROTL^{29}(x)$	$s_3(x) = SHR^2(x) \oplus SHL^2(x) \oplus ROTL^{28}(x) \oplus ROTL^{59}(x)$
$s_4(x) = SHR^1(x) \oplus x$	$s_4(x) = SHR^1(x) \oplus x$
$s_5(x) = SHR^2(x) \oplus x$	$s_5(x) = SHR^2(x) \oplus x$
$r_1(x) = ROTL^3(x)$	$r_1(x) = ROTL^5(x)$
$r_2(x) = ROTL^7(x)$	$r_2(x) = ROTL^{11}(x)$
$r_3(x) = ROTL^{13}(x)$	$r_3(x) = ROTL^{27}(x)$
$r_4(x) = ROTL^{16}(x)$	$r_4(x) = ROTL^{32}(x)$
$r_5(x) = ROTL^{19}(x)$	$r_5(x) = ROTL^{37}(x)$
$r_6(x) = ROTL^{23}(x)$	$r_6(x) = ROTL^{43}(x)$
$r_7(x) = ROTL^{27}(x)$	$r_7(x) = ROTL^{53}(x)$
$AddElement(j) = \Big(ROTL^{(j+1)}(M_j^{(i)}) + ROTL^{(j+4)}(M_{j+3}^{(i)}) - ROTL^{(j+11)}(M_{j+10}^{(i)}) + K_{j+16}\Big) \oplus H_{j+7}^{(i)}$	$AddElement(j) = \Big(ROTL^{(j+1)}(M_j^{(i)}) + ROTL^{(j+4)}(M_{j+3}^{(i)}) - ROTL^{(j+11)}(M_{j+10}^{(i)}) + K_{j+16}\Big) \oplus H_{j+7}^{(i)}$
$expand_1(j) = s_1(Q_{j-16}^{(i)}) + s_2(Q_{j-15}^{(i)}) + s_3(Q_{j-14}^{(i)}) + s_0(Q_{j-13}^{(i)}) + s_1(Q_{j-12}^{(i)}) + s_2(Q_{j-11}^{(i)}) + s_3(Q_{j-10}^{(i)}) + s_0(Q_{j-9}^{(i)}) + s_1(Q_{j-8}^{(i)}) + s_2(Q_{j-7}^{(i)}) + s_3(Q_{j-6}^{(i)}) + s_0(Q_{j-5}^{(i)}) + s_1(Q_{j-4}^{(i)}) + s_2(Q_{j-3}^{(i)}) + s_3(Q_{j-2}^{(i)}) + s_0(Q_{j-1}^{(i)}) + AddElement(j-16)$	$expand_1(j) = s_1(Q_{j-16}^{(i)}) + s_2(Q_{j-15}^{(i)}) + s_3(Q_{j-14}^{(i)}) + s_0(Q_{j-13}^{(i)}) + s_1(Q_{j-12}^{(i)}) + s_2(Q_{j-11}^{(i)}) + s_3(Q_{j-10}^{(i)}) + s_0(Q_{j-9}^{(i)}) + s_1(Q_{j-8}^{(i)}) + s_2(Q_{j-7}^{(i)}) + s_3(Q_{j-6}^{(i)}) + s_0(Q_{j-5}^{(i)}) + s_1(Q_{j-4}^{(i)}) + s_2(Q_{j-3}^{(i)}) + s_3(Q_{j-2}^{(i)}) + s_0(Q_{j-1}^{(i)}) + AddElement(j-16)$
$expand_2(j) = Q_{j-16}^{(i)} + r_1(Q_{j-15}^{(i)}) + Q_{j-14}^{(i)} + r_2(Q_{j-13}^{(i)}) + Q_{j-12}^{(i)} + r_3(Q_{j-11}^{(i)}) + Q_{j-10}^{(i)} + r_4(Q_{j-9}^{(i)}) + Q_{j-8}^{(i)} + r_5(Q_{j-7}^{(i)}) + Q_{j-6}^{(i)} + r_6(Q_{j-5}^{(i)}) + Q_{j-4}^{(i)} + r_7(Q_{j-3}^{(i)}) + s_4(Q_{j-2}^{(i)}) + s_5(Q_{j-1}^{(i)}) + AddElement(j-16)$	$expand_2(j) = Q_{j-16}^{(i)} + r_1(Q_{j-15}^{(i)}) + Q_{j-14}^{(i)} + r_2(Q_{j-13}^{(i)}) + Q_{j-12}^{(i)} + r_3(Q_{j-11}^{(i)}) + Q_{j-10}^{(i)} + r_4(Q_{j-9}^{(i)}) + Q_{j-8}^{(i)} + r_5(Q_{j-7}^{(i)}) + Q_{j-6}^{(i)} + r_6(Q_{j-5}^{(i)}) + Q_{j-4}^{(i)} + r_7(Q_{j-3}^{(i)}) + s_4(Q_{j-2}^{(i)}) + s_5(Q_{j-1}^{(i)}) + AddElement(j-16)$

Appendix B

Performance measurements

As we refered earlier, Daniel Bernstein and the VAMPIRE lab presented the measumenent of performance of various hash functions on different machines. Specifically, they show the results of SHA-3 candidates, working with all claimed versions of the finalists, and SHA-2 (both SHA-256 and SHA-512). So, the compelete list includes the following hashes with specified parameters[45]:

- BLAKE-32: BLAKE with 32-bit words, 10 rounds, and 256-bit output
- BLAKE-64: BLAKE with 64-bit words, 14 rounds, and 512-bit output
- BLAKE-256: BLAKE with 32-bit words, 14 rounds, and 256-bit output
- BLAKE-512: BLAKE with 64-bit words, 16 rounds, and 512-bit output
- Grøstl with 256-bit output; tweaked starting in supercop-20110426
- Grøstl with 512-bit output; tweaked starting in supercop-20110426
- JH with 224-bit output
- JH with 256-bit output
- JH with 384-bit output
- JH with 512-bit output

- Keccak[]=Keccak[r=1024,c=576,nr=24] with 1024-bit output; padding tweaked starting in supercop-20110426
- Keccak[r=1152,c=448,nr=24] with 224-bit output; padding tweaked starting in supercop-20110426
- Keccak[r=1088,c=512,nr=24] with 256-bit output; padding tweaked starting in supercop-20110426
- Keccak[r=832,c=768,nr=24] with 384-bit output; padding tweaked starting in supercop-20110426
- Keccak[r=576,c=1024,nr=24] with 512-bit output; padding tweaked starting in supercop-20110426
- Skein-256-256 with 256-bit output (tweaked starting in supercop-20100924)
- Skein-512-512 with 512-bit output (tweaked starting in supercop-20100924)
- Skein-1024-1024 with 1024-bit output (tweaked starting in supercop-20100924)

The test cases were done for long and short messages (4096, 1536, 576, 64 and 8 bytes) on more then 130 different architectures. One of the representations of the achieved results is the graph, which is the superimposition of several curves, one curve per each hash function. The graph is supported by the set of tables, where rows contain values of the speed measurements, one table for each lenght of the input message. The check points of each measurement are the first quartile, the median and the third quartile. The sample of such table you can find in the end of this appendix (for more details see [45]). According to the experiments, in particular each machine has 6 characterizing tables, but we just don't list them all here. Another representation of the results contains the speed values for all the machines on every function. This approach allows to make a comparison between the candidates and their different versions.

This work provides great statistical information and gives the feeling of the general speed of the functions among the final 5. It seems that BLAKE and Skein are the fastest ones on most of machines, which is not suprising, by the way, because one of the claim of the Schneiers' team was the high speed of Skein. At the same time we should say that Keccak does well enough also on different configurations, which allows it to stay in the middle. For more interesting results and comparisons, see [45]

Table 18. Sample of the experiment: Cycle/byte for long messages

quartile	median	quartile	hash
20.21	20.21	20.21	skein512
23.32	23.38	23.39	blake32
24.26	24.29	24.29	skein256
26.18	26.19	26.20	skein1024
26.46	26.52	26.58	blake64
26.81	26.81	26.84	blake256
29.83	29.86	29.89	blake512
33.24	33.33	33.43	keccakc448
35.49	35.53	35.57	keccakc512
37.80	37.81	37.83	sha512
38.27	38.29	38.31	keccak
47.72	47.73	47.98	keccak768
51.05	51.06	51.08	jh224
51.06	51.07	51.07	jh384
51.06	51.07	51.07	jh512
51.07	51.07	51.20	jh256
60.54	60.55	60.55	sha256
66.41	66.42	66.43	keccak1024
115.18	115.30	115.37	groestl256
182.04	182.21	182.28	groestl512

amd64, 1000MHz, Intel Atom N455 (106ca), h2atom, SUPERCORP version: 2011-04-29, benchmark dates: 2011-05-03 - 2011-05-14